UC Riverside

UC Riverside Electronic Theses and Dissertations

Title

Architectural Support for Securing Systems Against Software Vulnerabilities

Permalink

https://escholarship.org/uc/item/27k7m46p

Author

Khasawneh, Khaled N.

Publication Date

2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Architectural Support for Securing Systems Against Software Vulnerabilities

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Khaled Nofan Khasawneh

September 2019

Dissertation Committee:

Professor Nael Abu-Ghazaleh, Chairperson
Professor Laxmi Bhuyan
Professor Rajiv Gupta
Professor Zhiyun Qian

Copyright by
Khaled Nofan Khasawneh
2019

The Dissertation of Khaled Nofan Khasawneh is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Finishing up this dissertation mixes up my feelings; while I am certainly happy and excited to start a new chapter in my life, the past five years here at Riverside have been some of the most wonderful in my life, actually, if you ask me, I would prefer to never leave! There is a large number of people that I would like to thank individually for making this happens, and I do apologize in advance if I (inevitably) miss some :)

First of all, I would like to express my sincere gratitude to my research advisor, professor Nael Abu-Ghazaleh. He has supported me in my professional and personal lives by providing guidance and mentorship. His big enthusiasm and wealth of technical knowledge about research inspired and motivated me to keep forward to overcome the all difficulties and challenges. In the personal aspect, he always accommodated to my personal needs and gave me a lot of freedom. It is my great pleasure and honor to have known him.

I am also thankful to my dissertation committee members; professors Laxmi Bhuyan and Zhiyun Qian for providing me invaluable feedback. In addition to professor Rajiv Gupta whom was always generous in giving guidance regarding my professional development.

I also want to thank my collaborators, Dmitry Ponomarev, Lei Yu, Chengyu Song, Dmitry Evtyushkin and Daniel Townley. Without their collaborations, I would never finish and publish my work in the past few years. I also like to thank my colleagues in our group: Hoda Naghibijouybari, Fatemah Alharbi, Ahmed Abdo, Hodjat Asghari, Abdulrahman Bin Rabiah, Shafiur Rahman, Shirin HajiAminShirazi, Sakib Md Bin Malek, and specially Esmaeil (Reza) Mohammadian Koruyeh. I would never forget the time I have spent with them. Also, I would like to thank my friends: Ahamed

Alzoubi, Raed Shawabkah, Khawla Bawadi, Haifa Khrais, Noor Alshareef, Mohammad Mansi, Derar Issa, Hadi Zamani, and Khaled Abu Rajab who have always got my back. Without you, my PhD life won't be this fascinating.

It goes without saying, without support from my parents, Nofan Khasawneh, and Samar Al-Awamleh, this work is not possible. In addition to my brother Amro Khasawneh, and sisters, Rand Khasawneh and Roaa Khasawneh. They have tolerated my desire to be a student for seemingly as long as possible. Throughout my life, their love and support has enabled me to pursue what truly interests me and has made me the person I am today. Also, I would like to thank my cousins, who inspired me and supported me, since I was a child, to finish my graduate studies.

I would like to thank my in-laws, Abdel Muti Alsyouri, Salwa Bawadi, Zaid, Haneen, Hussam, and not to forget my best friend Rahaf Al-Sayouri (who was also incredibly supportive to help me through this process). I wouldnt have reached this far without their constant and unconditional love and support.

Finally, and also most importantly, the contributions of my wife, Saba Al-Sayouri, are invaluable; she sacrificed by doing her PhD remotely just to stay with me. In addition, to being wonderfully understanding during my dissertation and job hunting process. Without her support and her frequent reminders to stay sane and healthy, none of this work would have seen the light of the day.

Dedicated to my family, wife, in-laws, and grandparents

ABSTRACT OF THE DISSERTATION

Architectural Support for Securing Systems Against Software Vulnerabilities

by

Khaled Nofan Khasawneh

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2019
Professor Nael Abu-Ghazaleh, Chairperson

Cyberattacks are the fastest growing crime in the U.S., and they are increasing in size, sophistication, and cost. These attacks use vulnerabilities to compromise systems to leak Information (Yahoo 2016, Marriott 2018, and Facebook 2019), steal identity information (Equifax 2017), or even affecting politics (by attacking the governmental election process). Traditionally, security researchers and practitioners have viewed security as a software problem – originating in software and to be solved by software. Recently, the Spectre and Meltdown attacks have shown that hardware should also be considered when evaluating the system security. Conversely, because many aspects of security are computationally expensive, hardware can play a role in promoting software security through computational support as well as the development of new abstractions that promote security. Under this general umbrella, the research in this dissertation pursues two research directions that demonstrate how hardware can promote software security, and how we can design hardware that is secure against Spectre and Meltdown attacks.

In the first direction, security exploits and ensuant malware pose an increasing challenge to computing systems as the variety and complexity of attacks continue to increase. In response, software-based malware detection tools have grown in complexity,

thus making it computationally difficult to use them to protect systems in real-time. Against this drawback, hardware-based malware detectors (HMDs) are a promising new approach to defend against malware. HMDs collect low-level architectural features and use them to classify malware from normal programs. With simple hardware support, HMDs can be always on, operating as a first line of defense that prioritizes the application of more expensive and more accurate software-detector. In this dissertation, our goal is to make HMDs practical for deployment in two ways:

(1) Improving the detection accuracy of HMDs: We use specialized detectors targeted towards a specific type of malware to improve the detection of each type. Next, we use ensemble learning techniques to improve the overall accuracy by combining detectors. We explore detectors based on logistic regression (LR) and neural networks (NN). The proposed detectors reduce the false-positive rate by more than half compared to using a single detector, while increasing their sensitivity. We develop metrics to estimate detection overhead; the proposed detectors achieve more than 16.6x overhead reduction during online detection compared to an idealized software-only detector, with an 8x improvement in relative detection time. NN detectors outperform LR detectors in accuracy, overhead (by 40%), and time-to-detection of the hardware component (by 5x). Finally, we characterize the hardware complexity by extending an open-core and synthesizing it on an FPGA platform, showing that the overhead is minimal.

(2) Make them resilient to evasion attacks: we explore the question of how well evasive malware can avoid detection by HMDs. We show that existing HMDs can be effectively reverse-engineered and subsequently evaded, allowing malware to hide from detection without substantially slowing it down (which is important for certain types of malware). This result demonstrates that the current generation of HMDs can be easily defeated by evasive malware. Next, we explore how well a detector can evolve if it is

exposed to this evasive malware during training. We show that simple detectors, such as logistic regression, cannot detect the evasive malware even with retraining. More sophisticated detectors can be retrained to detect evasive malware, but the retrained detectors can be reverse-engineered and evaded again. To address these limitations, we propose a new type of Resilient HMDs (RHMDs) that stochastically switch between different detectors. These detectors can be shown to be provably more difficult to reverse engineer based on resent results in probably approximately correct (PAC) learnability theory. We show that indeed such detectors are resilient to both reverse engineering and evasion, and that the resilience increases with the number and diversity of the individual detectors. Our results demonstrate that these HMDs offer effective defense against evasive malware at low additional complexity.

In the second direction, the recent Spectre and Meltdown attacks show that speculative execution, which is used pervasively in modern CPUs, can leave side effects in the processor caches and other structures even when the speculated instructions do not commit and their direct effect is not visible. Therefore, they utilize this behavior to expose privileged information accessed speculatively to an unprivileged attacker. In particular, the attack forces the speculative execution of a code gadget that will carry out the illegal read, which eventually gets squashed, but which leaves a side-channel trail that can be used by the attacker to infer the value. Several attack variations are possible, allowing arbitrary exposure of the full kernel memory to an unprivileged attacker. In this dissertation, we introduce a new model (SafeSpec) for supporting speculation in a way that is immune to the side-channel leakage necessary for attacks such as Meltdown and Spectre. In particular, SafeSpec stores side effects of speculation in separate structures while the instructions are speculative. The speculative state is then either committed to the main CPU structures if the branch commits, or squashed if it does not, making

all direct side effects of speculative code invisible. The solution must also address the possibility of a covert channel from speculative instructions to committed instructions before these instructions are committed (i.e., while they share the speculative state). We show that SafeSpec prevents all three variants of Spectre and Meltdown, as well as new variants that we introduce. We also develop a cycle accurate model of modified design of an x86-64 processor and show that the performance impact is negligible (in fact a small performance improvement is achieved). We build prototypes of the hardware support in a hardware description language to show that the additional overhead is acceptable. SafeSpec completely closes this class of attacks, retaining the benefits of speculation, and is practical to implement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We live in an increasingly connected and autonomous world where security breaches to computing systems not only threaten our data but also our safety and livelihood. Critical infrastructure is increasingly interconnected and offers interfaces open to malicious actors. Cyberattacks are the fastest growing crime in the U.S., and they are increasing in size, sophistication, and cost [152] – even major companies such as Marriott [119], Equifax [44], Yahoo [102], and Facebook [66] find themselves the target of serious cyber attacks; the leaks of information from Yahoo in 2016, Marriott in 2018, and Facebook in 2019 directly affected 500M, 500M, and 540M users respectively [102, 119, 66]. Identity information, including the social security numbers, of 143M Equifax U.S. customer, were stolen in 2017 [44]. Furthermore, affecting even politics, specifically the governmental election process is not immune to cyber attacks [130]. Therefore, cybersecurity is a fundamental requirement that should be considered when designing and evaluating computing systems on par with performance, functionality, or power consumption.

Traditionally, security researchers and practitioners have viewed security as a software problem – originating in software and to be solved by software. However, in this dissertation, we argue that **hardware can be used to help defenses but also offers vulnerabilities**:

- On one hand, much attention has been directed to the study of security defenses at the system and application levels for many reasons, such as they are easier to deploy and to update. However, these defenses may incur high overhead to the system since it would consume hardware resources, which limit it's use. Therefore, hardware support can offer an advantage in terms of performance overhead by providing accelerators for them in the hardware.

- On the other hand, the recent Spectre and Meltdown attacks, which target speculative execution, a computer architecture technique used for performance, showed that vulnerabilities can arise from the architecture even if there are no weaknesses in the software.

Therefore, this dissertation pursues approaches to building defenses in the architecture in two directions: **(1) Hardware support for accelerating software defenses** and **(2) Hardware support for closing hardware vulnerabilities**. In the first direction, we were able to show that using the hardware support we can build not only accurate hardware malware detectors (HMDs) that offer performance advantage for the dynamic malware detection problem but also resilient to evasion attacks from attackers. In the second direction, we examine vulnerabilities in computer hardware that are exploitable by software and how to design systems that are not exposed by such vulnerabilities. In particular, we developed an architectural principled solution that can close all variants of Spectre and Meltdown.

In the remainder of this chapter, we will first describe the research problems this dissertation is trying to solve and the limitations of existing solutions in Section 1.1. Then, we will follow by describing the contributions of this dissertation in Section 1.2. Finally, we will describe the organization of this dissertation in Section 1.3.

1.1 Motivation: Limitations of State of the art

This dissertation pursues approaches to building defenses in the architecture in two directions: **(1) Hardware support for accelerating software defenses** and **(2) Hardware support for closing hardware vulnerabilities**. Figure 1.1 shows an overview of this dissertation; in the first direction, to demonstrating an important class of defenses that can be helped by hardware we choose malware detection. In particular, dynamic malware detection since it requires monitoring the programs at all times, which incur high overhead to the system that resulted in limiting it's use in practice. Against this drawback, this dissertation explores building practical hardware accelerators to solve the dynamic malware detection limitations.

On the second direction, to demonstrating an important hardware vulnerability we choose speculation attacks. Speculation attacks is a very dangerous class of attacks that showed that vulnerabilities can arise from the architecture even if there are no weaknesses in the software. Against this class of attacks, this dissertation explores developing an architectural principled solution to close this class of attacks.

In this section, we will describe both problems (malware detection and speculation attacks) that this dissertation is trying to solve and show the limitations of state of the art solutions.

1.1.1 Malware Detection

Malware detection is important since computing systems are under continuous attacks by increasingly motivated and sophisticated adversaries; for example, over 900 million malware sample were reported by AV TEST in their malware zoo, with over 50 million coming in the first half of 2019 [11]. These attackers use vulnerabilities to compromise systems and deploy malware (malicious software) [6]. Malware is created for profit through forced advertising (adware) [31], spreading illicit and contraband material or email spam (botnets) [137], stealing sensitive information (spyware) [26], or to extort money (ransomware) [62]. Although significant efforts continue to be directed at making systems more difficult to attack, the number of exploitable vulnerabilities is overwhelming, and attacks could be installed by fooling users through phishing and repackaged malware [178].

Since preventing malware from compromising the system is impossible, the second approach of defense is to detect malware while/before compromising the system. Traditionally, two techniques are used for malware detection: static malware detection and dynamic malware detection. Static malware detection looks for signatures in the executables. Example of static malware detection is antivirus software [166]. However, it can be easily defeated by using various code obfuscation techniques [178]. On the other hand, dynamic malware detection monitors the behavior of the running program to detect malicious activities. However, the complexity and difficulty of dynamic monitoring have traditionally limited its use due to constrained resources (10x slowdown is common) [43].

Against this backdrop, researchers (and some commercial products [132]) have been exploring the use of hardware malware detectors (HMD), which are always on and

which have little to no impact on performance and complexity, to protect computing systems. Specifically, HMDs are envisioned as the first line of defense to alert/prioritize the activities of more accurate but more expensive software detectors [124]. However, these detectors are not mature enough for practical deployment; (1) accuracy is not very high and (2) their resilience to adversarial attacks has not been studied. Therefore, this dissertation addresses the state of the art solutions limitations to bring HMDs closer for practical deployment. In particular, this dissertation explores ways of building more accurate HMDs using ensemble techniques and make them resilient to evasion attacks.

1.1.2 Speculation Attacks

Security and privacy are important due to the increasing amount of secret and sensitive data that share the same infrastructure. Sharing the infrastructure allow adversaries to steal information thought a side-channel [75, 115, 73]. Side channel attacks are one of the most dangerous vulnerabilities to the could since it exploits weaknesses in the implementation of otherwise secure systems and algorithms. Even isolated execution environments with enforcing strict access control are not immune to information leaks through hardware resources. The recent dangerous speculation attacks, such as Meltdown and Spectre [98, 89, 58, 106], target speculative execution (a standard microarchitectural technique used in virtually all modern CPUs to improve performance) to access sensitive data and leak it through a side-channel. Several attack variations have been demonstrated, including arbitrary exposure of the full memory of other processes, OS kernel, hypervisor, and even SGX enclaves [29] to an unprivileged attacker, making this a dangerous open attack vector on modern systems.

Although several defenses and software patches have been proposed to mitigate Spectre and Meltdown [159, 53], they often address only one aspect of the attack,

leaving attackers with other possible variations that are still available. Besides, these patches often lead to high overheads: 10-30% reported on average, but often much higher. For example, Netflix reported 800% slowdown with the Meltdown patches on their systems [158, 52].

1.2 Contributions of the Dissertation

The research in this dissertation focuses on showing that architecture support for security can help software be more secure by providing acceleration for software solutions that are not practical due to their high overhead or by re-designing the architecture to close software exploitable vulnerabilities that software solution would incur high overhead on the system, need recompilation of the programs, and/or need user input/feedback (example, marking the secret data).

Therefore, in this section, we describe our contributions towards solving the problems described in the previous section (Section 1.1); malware detection and speculation attacks.

1.2.1 Malware Detection

We explored two research questions within this space: **(1)** How to build accurate and inexpensive hardware detectors using ensemble learning techniques? and **(2)** how to make detectors resilient to evasion from attackers?

(1) Accurate Hardware Malware Detectors (HMDs)

Hardware-based malware detectors (HMDs) are a promising new approach to defend against malware. HMDs collect low-level architectural features and use them to classify malware from normal programs. With simple hardware support, HMDs can be

always on, operating as a first line of defense that prioritizes the application of more expensive and more accurate software-detector. In this project, our goal is to increase the accuracy of HMDs, to improve detection, and reduce overhead. We use specialized detectors targeted towards a specific type of malware to improve the detection of each type. Next, we use ensemble learning techniques to improve the overall accuracy by combining detectors. We explore detectors based on logistic regression (LR) and neural networks (NN). The proposed detectors reduce the false-positive rate by more than half compared to using a single detector, while increasing their sensitivity. We develop metrics to estimate detection overhead; the proposed detectors achieve more than 16.6x overhead reduction during online detection compared to an idealized software-only detector, with an 8x improvement in relative detection time. NN detectors outperform LR detectors in accuracy, overhead (by 40%), and time-to-detection of the hardware component (by 5x). Finally, we characterize the hardware complexity by extending an open-core and synthesizing it on an FPGA platform, showing that the overhead is minimal.

(2) Evasion-Resilient Hardware Malware Detectors (HMDs)

Several aspects of the HMDs construction have been explored, leading to detectors with high accuracy. However, in this project, we explore the question of how well evasive malware can avoid detection by HMDs. We show that existing HMDs can be effectively reverse-engineered and subsequently evaded, allowing malware to hide from detection without substantially slowing it down (which is important for certain types of malware). This result demonstrates that the current generation of HMDs can be easily defeated by evasive malware. Next, we explore how well a detector can evolve if it is exposed to this evasive malware during training. We show that simple detectors, such as

logistic regression, cannot detect the evasive malware even with retraining. More sophisticated detectors can be retrained to detect evasive malware, but the retrained detectors can be reverse-engineered and evaded again. To address these limitations, we propose a new type of Resilient HMDs (RHMDs) that stochastically switch between different detectors. These detectors can be shown to be provably more resilient to evasion based on resent results in probably approximately correct (PAC) learnability theory. We show that indeed such detectors are resilient to both reverse engineering and evasion, and that the resilience increases with the number and diversity of the individual detectors. Our results demonstrate that these HMDs offer effective defense against evasive malware at low additional complexity.

1.2.2 Speculation Attacks

We explored building a principled solution for speculation attacks that can be used to protect against all variants of attacks with minimum impact on performance.

Leakage Free Speculation

Speculative attacks, such as Spectre and Meltdown, target speculative execution to access privileged data and leak it through a side-channel. To solve this problem, we introduce (*SafeSpec*), a new model for supporting speculation in a way that is immune to the side-channel leakage by storing side effects of speculative instructions in separate structures until they commit. Additionally, we address the possibility of a covert channel from speculative instructions to committed instructions before these instructions are committed. We develop a cycle accurate model of modified design of an x86-64 processor and show that the performance impact is negligible.

1.3 Organization of This Dissertation

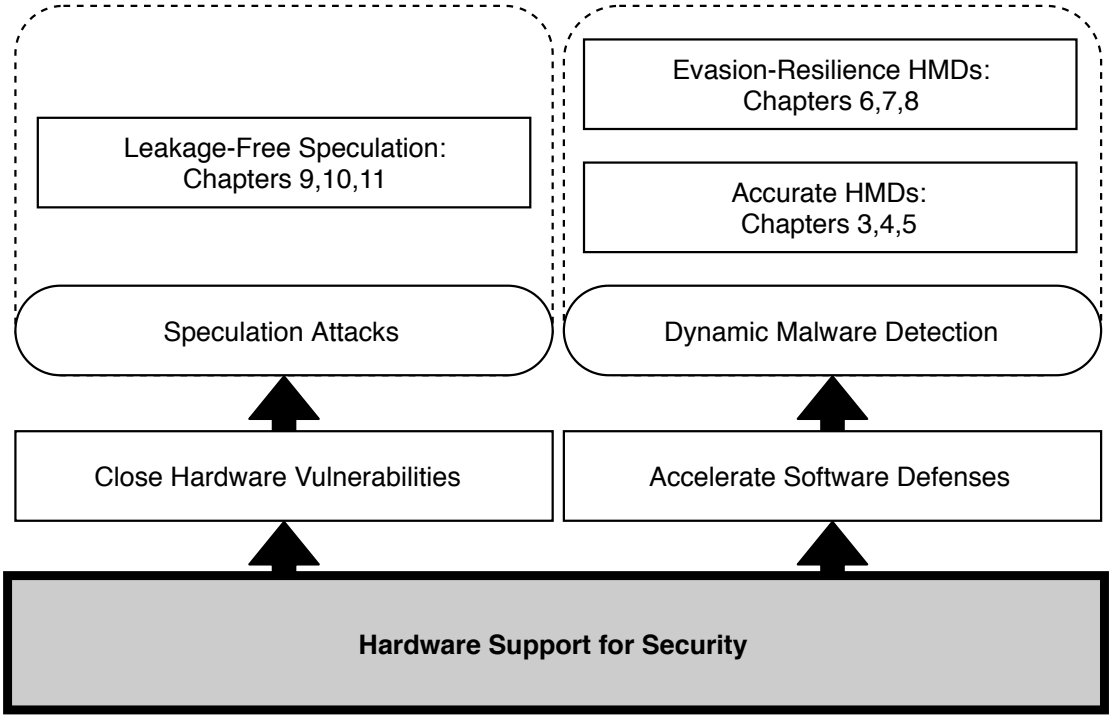


Figure 1.1: Dissertation overview

The goal of this dissertation is to show the importance of hardware support in security; either through providing acceleration for software defenses or through closing vulnerabilities that originate from the hardware. Therefore, in Chapter 2, we will describe the related works for the example problems that are used to support the goal of this dissertation; malware detection and speculation attacks. The rest of this dissertation will be organized into two main parts as follows:

PART 1: Hardware support for accelerating software defenses

This part is about exploring acceleration solutions for software defenses through hardware support. Therefore, to solve the high overhead of the dynamic malware detection solution through hardware support and make the hardware support for malware detection more practical for deployment, this part can be divided into two main projects

(1) Accurate HMDs: building accurate and inexpensive hardware detectors using ensemble learning techniques **(2) Evasion-Resilience HMDs:** building HMDs that are resilient to evasion attacks. The organization of the two projects is as follows:

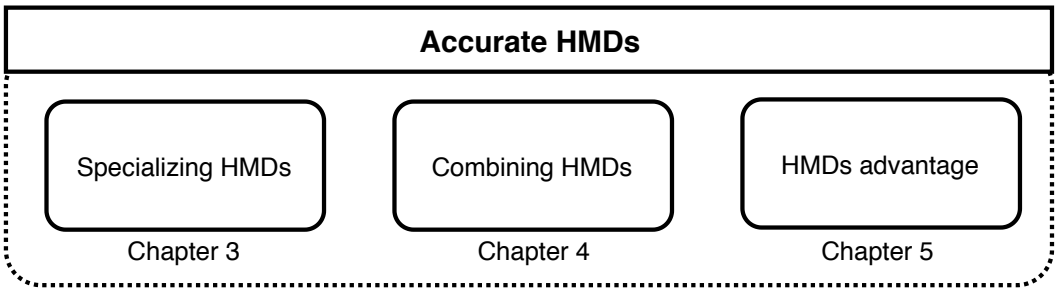


Figure 1.2: Building accurate HMDs project overview

(1) Accurate HMDS: in Chapter 3, we investigate whether specialized HMDs, each targeting a specific type of malware, can more successfully classify that type of malware. In Chapter 4, we combine multiple detectors, general or specialized, to improve the overall performance of the detection and we analyze the implications on the hardware complexity of the different configurations. In Chapter 5, we develop metrics that translate detection performance of HMDs to overhead and time-to-detection advantages of the whole system and we conducted a longitudinal study to explore whether detectors trained on a set of malware continue to be effective over time as malware evolves. Figure 1.2 shows an overview of the chapters that compose this project.

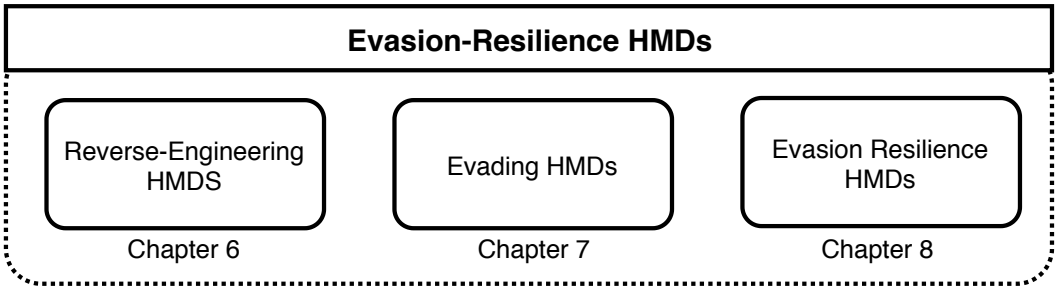


Figure 1.3: Building evasion-resilience HMDs project overview

(2) Evasion-Resilience HMDS: in chapter 6, we describe our threat model and investigate wither HMDs be reverse-engineered. In Chapter 7, we explore wither having a model of the detector, can malware developers modify malware to avoid detection. Then we study the ability of malware to evade detection even if the detector is re-trained with some samples of the evasive malware. In Chapter 8, we explore whether new HMDs can be constructed that are robust to evasion and if they fundamentally increase the difficulty of evasion or simply present another hurdle that can be bypassed by attackers. Figure 1.3 shows an overview of the chapters that compose this project.

PART 2: Hardware support for closing hardware vulnerabilities

This part is about closing vulnerabilities that originate from the hardware through hardware support. Therefore, we selected one of the most dangerous class of attacks that exploit a hardware vulnerability (speculation attacks) to defend against through hardware support. The organization of this project is as follows:

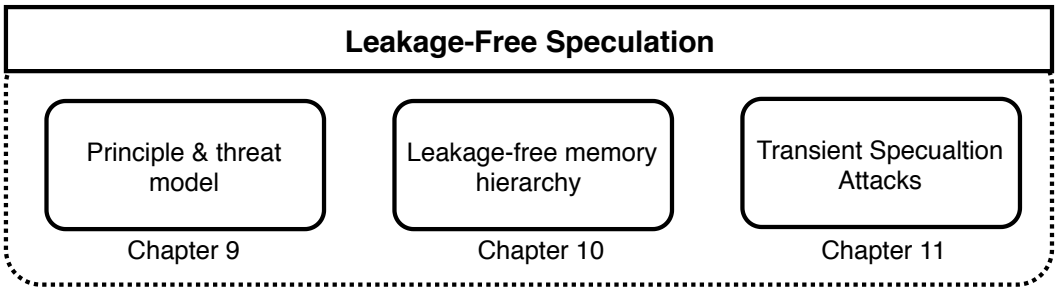


Figure 1.4: Leakage-free speculation project overview

Leakage-Free Speculation: to defend against speculation attacks, in Chapter 9, we present the background needed and describe our principled solution (SafeSpec). In Chapter 10, we describe the design of building a memory hierarchy (caches and TLBs) that are free from speculation-induced leakage and presents a performance, complexity, and security analysis of SafeSpec. In Chapter 11, we show that a covert channel can

be created as a byproduct of applying SafeSpec (we call this type of attacks *transient speculation attacks* (TSAs)) and explore how to construct the shadow state to mitigate TSAs. Figure 1.4 shows an overview of the chapters that compose this project.

Finally, Chapter 6 concludes before highlighting potential future work.

Chapter 2

Related Work

In this chapter, we present the related work of this dissertation. The organization of this chapter will be as follows: Section 2.1 will present the related work in the malware detection space, hardware malware detection, and adversarial machine learning. In Section 2.2, we will review all variants of speculation attacks. Then, we will continue to present the state of the art defense that where proposed to protect against speculation attacks.

2.1 Malware Detection

In this section, we discuss related work organized into two main parts. First, we review related work in malware detection. In the second part, we discuss adversarial classification and some important recent results in that domain.

2.1.1 Hardware Malware Detection

Malware detection is an area that has attracted extensive research and commercial interest over the past decade. In general, malware detection techniques are either static (focusing on the structure of a program or system) or dynamic (analyzing the behavior during execution) [64].

Static approaches including virus and spyware scanners are the first line of defense in malware detection. Originally, these scanners operated using pattern matching to look for signatures of known malware. However, these approaches can be easily evaded using program obfuscation or simple code transformations that preserve the function of the malware but make it not match the patterns known to the scanner [114]. More advanced detectors based on semantic signatures have been proposed, and significantly improved the performance of static scanners [32]. Static approaches are limited and can be bypassed by sophisticated attackers [111]. In particular, code obfuscation techniques (polymorphic malware), and malware encryption (packing or metamorphic malware) are both sufficient to hide even from these more advanced detectors [111].

On the other hand, dynamic detection approaches observe the behavior of the program (or the system) as it runs and interacts with the environment. Dynamic behavior-based detection attempts to detect deviations from normal behavior of a program as it operates. It detects anomalies in the observed behavior compared to its model of normal behavior, which is often program-specific, to identify malware. A large number of software malware detectors have been investigated that vary in terms of the monitored events, the normal behavior model, and the detection algorithm [57, 135, 71, 64, 91]. The advantage of dynamic detection is that it is resilient to metamorphic and polymor-

phic malware [111, 107]; it can even detect previously unknown malware. However, the disadvantages include a typically high false positive rate, and the high cost of monitoring during run-time. Moreover, since detection is a one time (or periodic) process, malware can evade detection either probabilistically or by recognizing that it is being observed and acting normally for that period.

Hardware based Malware Detectors (HMD) have attracted significant interest recently because they can be always on and essentially free in terms of performance and resource impact. Bilar et al. were the first to use the frequency of opcodes occurrence in a program as a feature for discriminating normal programs from malware [18]. Santos et al. and Yan et al. use opcode sequence signatures as a feature [139, 172]. Runwal et al. use similarity graphs of opcode sequences [136]. Recently, Demme et al. suggested using features based on performance counters [37]. Tang et al. conducted a similar study but used unsupervised learning [153]. Kazdagli et al. [77] introduced several new improvements to the construction methodology for both supervised and unsupervised learning based HMDs applied to mobile malware. All of these studies conduct offline analysis; none of the studies explore online detection or implementation using hardware. On the other hand, similar to our work, a real-time hardware malware detector was built by Ozsoy et al. [122]. However, they used a single detectors while our work uses an ensemble of specialized detectors.

Chavis et al. proposed an enterprise-class antivirus analysis framework, called SNIFFER [27]. Similar to our work, each machine in the network collect low-level features online using hardware. For each detection period, each machine transfers its collected features securely to a server which classifies them (hardware feature collection, but software classification). The detectors are general detectors; we believe that the ensemble techniques that improve the accuracy can reduce the false positives in this

system as well. Both the settings and the malware types (network attacks) considered by the paper are different than our environment.

Ensemble learning is a well-known technique in machine learning to combine the decisions of multiple base detectors to improve accuracy [169]. Ensemble learning is attractive because of its generalization ability which is much powerful than using one learner [39]. For ensemble detector to work, the base detectors have to be diverse; if the detectors are highly correlated, there is little additional value from combining them [134]. In this work, the diversity is based on different features (general ensemble detector), data sets (mixed ensemble detector), or both (specialized ensemble detector). In contrast to traditional machine learning approaches that use the training data to learn one hypothesis, some of our ensembles learn a set of subset-hypotheses (specialized detectors) and combine them. Ensemble learning may be considered a form of two-stage detection with the base detectors in the first stage, and the synthesis of their decision in the second. A two stage anomaly detector was proposed by Zhang et al. [181, 182] who use machine learning classifiers to build network traffic dependency graph, and then they used a root-trigger policy to identify outlier network requests. These works focus on the dependency knowledge of the first level, while our work focuses on combining the results of detectors that are trying to answer different questions.

The specialized ensemble detector combines multiple specialized detectors and dynamically collects the features to perform online detection. Researchers built ensemble malware detectors [180, 101, 175, 10, 116, 45, 177, 103, 145, 146, 61, 121, 49, 128], based on combining general detectors. Moreover, most of them used off-line analysis [175, 10, 145, 146, 45, 177, 61]. A few used dynamic analysis [116, 49, 128] and some used both static and dynamic analysis [101, 103, 121]. None of these works uses architecture features or is targeted towards hardware implementation (which requires simpler

machine learning algorithms). Specialized detectors were previously proposed [92] for use in malware classification (i.e., labeling malware). Labeling is used to classify collected malware using offline analysis which is a different application than the one we consider.

We present a few recent examples of the use of ensemble learning for malware detection in more detail. Zhang et al. [180] extracted n-gram features from the programs binary code and used it in malware detection. The ensemble composed of multiple probabilistic neural network (PNN) classifiers and the Dempster Shafer theory was utilized to combine them. Sami et al. [138] used API calls extracted from the Portable Executable (PE) Import Address table to build an ensemble detector using random forests. Mehmet et al. [121] created an ensemble detector for android malware by creating base detectors based on different features and learning algorithms. After that, the base detectors were combined using stacking or majority voting to form the ensemble system. However, the previous work does not try to combine specialized detectors to build the ensemble system. Smutz et al. recently explored the use of a diversified ensemble detector to classify possibly evasive PDF malware [149]. Note that all the above techniques use software detectors, on rich features and using advanced machine learning algorithms.

This work extends our prior work [82] which considered ensembles of only LR detectors. The NN ensemble detectors presented in this work are over 20% faster than the best LR ensemble detector while requiring 40% less overhead. Additionally, we study the speed of hardware component in detecting malware online and show that NN ensemble is 5x faster than the LR ensemble detector, potentially limiting the damage of the malware. This work also presents a hardware design of the NN ensemble detectors and analyzes their complexity. The malware longitudinal study in Section 5.2 is also a new contribution of this project, and demonstrates the need for HMDs to be retrainable.

Zhou et al., argue against the practicality of using hardware performance counters (HPCs) for malware detection [184]. They categorize the drawbacks that they observed into the following classes:

1. Dynamic Binary Instrumentation (DBI).

2. Virtual Machines (VMs).

3. No Cross-Validations or Insufficient Validations

The first class of drawbacks is using Dynamic Binary Instrumentation (DBI). DBI can be used to instrument the monitored program in order to collect features during it's execution. Examples of these tools are Pin [104], QEMU [13], Valgrind [117], or DynamoRIO [20]. The reason DBI is a drawback is that it introduces a substantial amount of performance overhead and is thus not suited to run in an always-on, online protection setting, which is the default use-case for current anti-malware suites. However, in our work we use DBI to collect features for two main reasons: (1) DBI feature collection is much more accurate and stable than HPCs [36], which results in more stable and accurate results (2) DBI was used just for experiments and identifying the features. However, as we show in Chapter 4.2, we extended the processor implementation by integrated a feature collection unit to the end of the commit stage of the processor's pipeline eliminate the overhead from using DBI for features collection.

The second class of drawbacks is using virtual machines (VMs) to run the monitored programs. The reason for using VMs in a drawback is that HPCs are limited and shared resource between the host and all VMs. Thus, virtualizing HPCs is a challenge in itself [36], which results in inaccurate HPCs values. However, using DBI to instrument programs inside a VM does not affect the results of the DBI.

The third drawback class is not using cross-validation in the experiments. Cross-validation [7] is used to prevent the machine learning model from overfitting [55] by examining the model with different inputs of training-and-testing examples. However, in our experiments, we used 10-folds cross-validation to make sure that we have made sufficient validation of the models.

Another issue of using HPCs for malware detection is the adversary arability to create evasive samples that can evade the detection [36]. Therefore, in our following work [79] (Chapters 6,7,8), we explored how malware writers may attempt to evade HMDs. The work shows that NN detectors are amenable to retraining when malware evolves. The work proposed creating multiple diverse detectors and switching between them randomly, which makes HMDs provably more robust to evasion attacks. While such detectors use multiple base detectors they use only one at a time. It is interesting to explore the combination of ensemble and evasion resilient HMDs.

2.1.2 Adversarial Machine Learning

Several studies have looked at attacking machine learning models. Attacks can be classified into two types: poisoning and evasion attacks [12]. In poisoning attacks, the adversary focuses on injecting malicious samples in the training data as an attempt to influence the accuracy of the model [17, 87]. For evasion attacks, similar to our own, the adversary crafts input samples that aim to be misclassified by the model [171, 96, 15, 50, 126]. Several evasion attacks were studied in the image classification field. An adversary can make changes to the pixels of an image to cause miss-classification of the image but will not change the visibility of the image to the human eye [15, 50, 126]. Since images have high entropy they can be easily manipulated

without changing the appearance of the image. On the other hand, in the malware detection domain, manipulating malware programs have different challenges since the functionality of the malware needs to be preserved. Evasion attacks in contexts outside image classification have also been considered. Such attacks are called *mimicry* attacks [21, 165]. Although recent studies [16, 35, 162] have provided theoretical grounds for randomization as a possible solution in adversarial classification, practical algorithms have yet to be developed for this problem.

Related to our evasion attack on malware detectors, researchers recently proposed evasive attacks on PDF malware detectors [171, 96]. These works consider static classifiers using structural features present in the PDF image. In contrast, our contribution targets detectors for a wide range of malware and we consider run-time anomaly detection using microarchitectural features. Besides the different nature of the application, our work makes a number of contributions relative to these recent works including showing how to reverse engineer the classifiers, reverse-engineering driven instruction injection to evade detection (they use random modifications), exploring the impact of retraining, and providing theoretical insights based on PAC theory into the structure of the problem. Moreover, these studies do not explore resilient classification.

Similar to the reverse engineering component of our work, Tramèr et al. [155] were able to reverse-engineer machine learning models against production Machine Learning-as-a-service (MLaaS) providers. However, they assumed that they know the features used by the target classifier. In addition, Shokri et al. [148] were able to use reverse-engineered models to perform a membership inference attack (given a data record and black-box access to a model, determine if the record was in the model's training dataset) against MLaaS providers. In both works, they attempt reverse engineering using random noise [155, 148]. We believe that this approach does not work in our threat model where

we do not have access to the classifier confidence (we see the output as a label, not as a probability), and where classification is a continuous process, which makes it difficult to assess incremental changes.

Khasawneh et al. used ensemble learning to improve the accuracy of HMDs [83, 78, 97]. Superficially, ensemble learning is similar to RHMD since it combines the output of multiple diverse detectors through a combiner function such as majority voting to improve the overall detection performance. However, since ensemble classifiers are deterministic, they can be reverse engineered and evaded. In contrast, the *stochastic switching* between individual detectors in RHMD makes both reverse-engineering and evasion difficult with a difficulty that increases with the number and diversity of the individual detectors. Smutz et al. also studied the use of an ensemble for PDF malware detection [149]; when the baseline detectors disagree, they consider this a possible indicator of evasive malware.

Although hardware-supported malware detection offers many advantages as it can be always on and has a low overhead on both power and performance, if HMDs are widely deployed, we must expect that attackers will attempt to evade detection as is the case in any adversarial setting. In this work, we show that without hardening HMDs, it is possible to reverse engineer and evade them, bringing into question the effectiveness of HMDs. Our work also explores whether retraining can be used to continue to track malware evasion, as well as the construction of resilient hardware malware detectors. With these results we believe evasion resilient HMDs become practical, bringing such solutions closer to practical deployment.

2.2 Speculation Attacks and Defenses

In this section, we discuss related work to speculation attacks organized into two main parts. First, we review variants of speculation attacks. In the second part, we discuss deployed and proposed defenses against speculation attacks.

2.2.1 Speculation Attacks

Since the initial announcement of Spectre and Meltdown in January of 2018, multiple variants of Spectre [90, 86, 93, 105, 48, 59, 29, 143] and Meltdown [99, 156, 160, 141, 161, 110, 140] have been proposed. Spectre attacks are characterized by manipulating the prediction mechanisms to trigger speculation to an attacker chosen gadget. They differ in what they exploit to trigger speculation: branch direction predictor (variant 1, variant 1.1) [90, 48, 86], branch target predictor (or branch target buffer) for variant 2 [90], return stack buffer for Spectre-RSB (also called variant 5) [93, 105], or load-store aliasing predictor for variant 4 [59].

On the other hand, Meltdown attack exploits speculative out-of-order instructions that lead to an exception. Multiple variants have been proposed: the original Meltdown [99], Foreshadow [160, 167], and RIDL [161] exploit the *page fault*, Meltdown variant 1.2 [86] exploit the *bound range exceeded exception*, Lazy FP [151] exploit the *device-not-available exception*, Meltdown variant 3a [9] exploit the *general protection fault*, Fallout [110] exploit the *write transient forwarding*, and ZombieLoad [141] exploit the *fill-buffer logic*. Canella et al. summarize these and additional variants [23].

To mitigate these attacks, several software and hardware defenses ranging from programming guidelines for cryptographic software developers [24] to architec-

tural changes [81, 173] have been proposed. In general, these solutions are ad hoc, often focused on a specific attack. Moreover, most result in substantial performance impact. In this section, we will overview these defenses categorized into three categories (1) speculation prevention: prevent speculation execution, (2) Secret data protection: provides secret data isolation, (3) Side-channel prevention: that interfere with side channel communication.

2.2.2 Speculation prevention

These defenses focus on preventing speculation by preventing misprediction [67, 120, 159, 68, 86] or faults [160]. Intel, AMD, and ARM proposed to use instructions that serialize the execution (e.g. `lfence`) to stop speculation around branches (e.g. both directions of the branch) [67, 8, 2]. Although liberal serialization (e.g., at every branch instruction) can mitigate some Spectre attacks, doing so severely hurts performance [67]: serializing all branch instructions will eliminate the performance benefit of the branch predictor (e.g., up to 10x slowdown [120]). Against this drawback, multiple proposals tried to reduce the number of serializing instructions introduced using static analysis to serialize execution around exploitable gadgets only [67, 108, 65, 164]. However, these approaches miss some of the gadgets that can be exploited [88]. Another weakness about these defenses is that even though they stop speculative execution around exploitable gadgets, they do not stop speculative code fetches and other micro-architectural behaviors before execution (e.g., instruction cache and iTLB fills) which can still leak data [143].

Furthermore, Speculative Load Hardening (SLH) [25] and You Shall Not Bypass (YSNB) [120] tried to reduce the high overhead of using liberal fences. Therefore, they proposed to identify Spectre gadgets, then injecting artificial dependencies between

branches and identified gadgets. Doing so will reduce the speculation window of the attack. Although this would results in performance advantage over liberal fencing, they still have 36%-60% performance overhead [154].

Google proposed Return Trampoline (retpoline) [159] as a software mitigation technique that defends against Spectre-BTB by replacing indirect branches with push+return instruction sequence that prevent BTB poisoning. However, this solution has high performance overhead since it stops speculation (similar to serialization). In addition, it can be bypassed using *ret* instructions since they cause miss-speculation through BTB; this is a by-product of a feature on Intel's Skylake+ processors (starting from Skylake) that allow processors to predict the address of a *ret* instruction from BTB in case of RSB underfilling. To solve this exploit, RSB stuffing [69] was proposed to intentionally fill the RSB with benign delay gadgets to avoid misspeculation on context switches. Although this technique can partially mitigate Spectre-BTB (when using *ret* to trigger speculation through BTB), it can also defend against SpectreRSB cross-domains attack. However, since we are filling the RSB on context switch, stored entries for the currently running process will be lost when execution is switched back to the current process (i.e. performance loss due to losing speculation information). Against this drawback, saves committed RSB entries per process in case of a context switch out of the process and restores them when execution returns to the process, which results in improving the prediction performance of *ret* instructions.

Intel and AMD added new instructions to their instruction set architecture (ISA) that can control indirect branches to defend against Spectre-BTB [70, 2]. The addition consists of three controls:

- Indirect Branch Restricted Speculation (IBRS): allows processors to enter IBRS mode (privileged mode) and execute indirect branches that are not influenced by less privileged mode.
- Single Thread Indirect Branch Prediction (STIBP): will not allow a hyperthread running on a core to use branch predictor entries inserted by the other thread running on the same core.
- The Indirect Branch Predictor Barrier (IBPB): allows processors to flush BTB and clear their state. This way the code executed before the barrier cannot impact branch prediction of the code executed after this instruction.

These new ISA instructions defend only against Spectre-BTB. In addition, they have a high performance overhead; up to 24% on Skylake and up to 53% on Haswell [41].

Moreover, SLoth [86] is a group of micro-architectural defenses that constrain store-to-load forwarding to defend against Spectre v1.1 and v4; (1) SLoth Bear: microcode update that prevents store-to-load forwarding from either speculative stores or to speculative loads. (2) SLoth: choose candidates for forwarding based on compiler marking instructions. (3) Arctic SLoth: apply dynamic detection of load and store pairs to determine candidates for forwarding. Nevertheless, this approach does not defend against Spectre v1 and require software, compiler, and hardware changes that result in performance overhead and implementation complexity.

An attractive hardware solution is Context-Sensitive Fencing (CSF) [154]. CSF is a micro-code mitigation technique were serialization instructions are added dynamically based on run-time conditions that identify potential exploit execution. Injecting serialization instructions dynamically reduces the impact of stopping speculation on performance which results in low performance overhead. Moreover, CSF proposed to

defend against Spectre v2 and Spectre-RSB using a special fence that would flush the BTB/RSB when transferring control to higher domains. However, flushing BTB and RSB would hurt performance since it will result in more mis-predictions. In addition, in simultaneous multithreading (SMT) processor, flushing the BTB/RSB after control transfer is not enough to protect against Spectre-BTB and Spectre-RSB since they can be polluted after the control transfer using other threads.

ConTExT [142] introduced protecting secret data from speculative execution. Basically, they proposed a new memory mapping (called non-transient mapping) which indicates data that must not be accessed by speculative instructions. Nevertheless, this solution requires changes to the architecture and the operating system, the developer involvement by annotating the secret data, and incur high performance overhead for security-critical applications.

2.2.3 Secret data protection:

These defenses focus on making sure that secret data can not be reached [53, 99]. However, they have limitations: Kernel Page-Table Isolation (KPTI) [53, 99] have performance overhead and some privileged memory locations must always remain mapped in user space due to x86 design [23], and Site Isolation [131] limits the amount of data that is exposed to side-channel attacks but attacks are still possible.

2.2.4 Side-channel prevention:

Since speculation attacks use a side-channel to communicate secret data out of the speculative execution gadget, these type of defenses is designed to prevent such communication. The benefits from such defenses are: (1) performance since they allow speculation (2) defend against all attack variants since all variants use side-channels to

leak the secret data. Furthermore, this category of defense is most relevant to our work since SafeSpec falls into this category.

Dynamically Allocated Way Guard (DAWG): DAWG [85] is a method to securely partition the cache at the cache way granularity to provide isolation between protection domains. Therefore, it requires changes to the cache and coherence protocol. In addition, it requires domains enforcement management in software. While this solution, similar to our defense, prevents leaking the data through a side-channel, it only protects across isolation domains and not those performed within the same address space or isolation domain.

InvisiSpec: most relevant to our work, and developed concurrently with it (SafeSpec technical report was disclosed in June, 2018 [81]), is an architectural solution called InvisiSpec [173]. Like SafeSpec, InvisiSpec is designed to make transient loads invisible in the cache hierarchy. InvisiSpec focus is on cache coherence and memory consistency rather than understanding the implications on a single core. They did not consider transient side channels on the shadow structures, sizing issues, or carry out overhead characterization. Moreover, InvisiSpec was focusing on protecting the d-cache while we developed attacks and defenses on i-cache and the TLB, applying the principle more widely.

Chapter 3

Specialized Hardware Mawlare Detectors

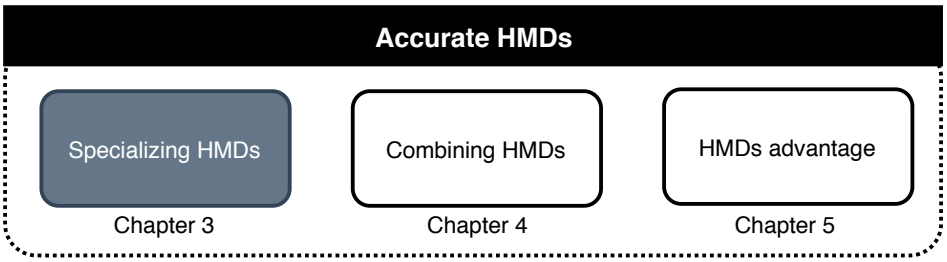


Figure 3.1: Building accurate HMDs project overview

Computing systems at all scales face a significant threat from malware; over 900 million malware sample was reported by AV TEST in their malware zoo, with over 50 million coming in the first half of 2019 [11]. Obfuscation and evasion techniques increase the difficulty of detecting malware after a machine is infected [178]. Zero-day exploits – novel exploits never seen before – defy signature based static analysis since their signatures have not been yet encountered in the wild. Thus, dynamic detection techniques [43] that can detect the malicious behavior during execution are needed [33, 94], to protect against such attacks. However, the difficulty and complexity of dynamic monitoring in software have traditionally limited its use.

Recent studies have shown that Hardware Malware Detectors (HMDs) that carry out anomaly detection in low-level feature spaces such as hardware events, can distinguish malware from normal programs [37, 18]. Then, an online HMD was introduced as hardware supported classifier trained using supervised learning that continuously monitors and differentiates malware from normal programs while the programs run [123, 124]. To tolerate false positives, this system is envisioned as a first step in malware detection to prioritize which processes should be dynamically monitored using a more sophisticated but more expensive second level of protection.

In this chapter, we pursue approaches to enhance the classification accuracy of HMDs. Improving accuracy increases their ability to detect malware, and reduces the overhead that results from false positives. Therefore, in this chapter, as shown is Figure 3.1, we explore whether specialized detectors, each targeting a specific type of malware, can more successfully classify that type of malware. After confirming that specialized detectors perform better than general detectors, we identify the features that perform best for each specialized detector. We characterize how specialized detectors trained for specific malware types perform compared to a general detector and show that specialization has significant performance advantages.

3.1 Approach and Evaluation Methodology

We consider a system with a hardware malware detector (HMD) similar to those recently proposed in literature [37, 123, 77]. HMD exploit the fact that the computational footprint of malware differs from that of normal programs in low-level feature spaces. Detectors built using such features appear to be quite successful; Qualcomm announced the use of a similar technology in their Snapdragon processor [133]. Early

studies relied on opcode mixes [18, 139, 172, 136]. More recently, Demme et al. [37] showed that malware programs can be classified effectively by the use of offline machine learning model applied to low-level features; in this case they used features available through hardware performance counters of the ARM processor collected periodically.

This work improves on prior work by Ozsoy et al. [123] who built an *online* hardware-supported, low-complexity, malware detector. The online detection problem uses a time-series window based averaging to detect transient malware behavior. As detection is implemented in hardware, simple machine learning algorithms are used to avoid the overhead of complex algorithms. This work demonstrated that low-level architectural features can be used to detect malware in real-time.

In this study, our goal is to improve the effectiveness of online HMD. Improving the detection performance leads to more malware being detected with fewer false positives. We explore using specialized detectors for different malware types to improve detection. We show that specialized detectors are more effective than general detectors in classifying their malware type. Furthermore, in Chapter 4, we study different approaches for combining the decisions of multiple detectors to achieve better classification. In this section, we present some details of the methodology including the Data Set and the choice of features for classification.

3.1.1 Data Set

Our data set consists of 3,653 malware programs and 554 regular Windows programs (the malware samples that we use are Windows-based). This regular program set contains the SPEC 2006 benchmarks [56], Windows system binaries, and many popular applications such as Acrobat Reader, Notepad++, and Winrar. The malware programs were chosen from the MalwareDB malware set [118].

The group of regular and malware programs were all executed within a virtual machine running a 32-bit Windows 7 with the firewall and security services for Windows disabled. We observed that this desktop malware does not require user interaction to operate maliciously, in contrast to prior work that showed that mobile malware does not run correctly without user interaction [77]. We verified that a large sample (more than half) of our malware ran correctly by manually checking run-time behaviour. In fact, the intrusion detection monitoring systems on our network were tripped several times due to malware trying to search for and attack other machines. Eventually, we set up the environment in an independent subnet. However, for the regular programs, we manually interacted with them to trigger an expressive representation. The Pin instrumentation tool [34] was used to gather the dynamic traces of programs as they were executed. Each trace was collected after 150 system calls for a duration of 5,000 system calls or 15 million committed instructions, whichever is first.

The malware data set consists of five types of malware:

1. **Backdoors**: bypass the normal authentication of the system.
2. **Password Stealers (PWS)**: steals user credentials using a key-logger and sends them along with the visited website to the attacker.
3. **Rogues**: pretend to be an antivirus program and try to sell the victim its services.
4. **Trojans**: appear to be harmless programs but contain malicious code.
5. **Worms**: attempt to spread to other machines using various methods.

We selected only malware programs that were labeled as malware by Microsoft and used the Microsoft classification for their type [109]. Each malware set (corresponding to the malware type) and the regular programs set were randomly divided into three

subsets; training (60%), testing (20%) and validation (20%) as shown in Table 3.1. These are typical ratios used in training classifiers. The training and testing sets were used to train and test the detectors respectively. The validation set was used for exploring the settings of training and detection.

We note that both the number of programs and the duration of the profiling of each program is limited by the computational and storage overheads; since we are collecting dynamic profiling information through Pin [34] within a virtual machine, collection requires several weeks of execution on a small cluster, and produces several terabytes of compressed profiling data. Training and testing is also extremely computationally intensive. This dataset is sufficiently large to establish the feasibility and provide a reasonable evaluation of the proposed approach.

Table 3.1: Malware data set breakdown; showing the number of samples that is used for training, testing, and validation from each malware type.

	Total	Training	Testing	Validation
Backdoor	815	489	163	163
Rogue	685	411	137	137
PWS	557	335	111	111
Trojan	1123	673	225	225
Worm	473	283	95	95
Regular	554	332	111	111

3.1.2 Feature Selection

At the architecture/hardware level, there are many features that could be collected. To enable direct comparison of the proposed ensemble detector against a single detector, we use the same features used by Ozsoy et al. [123]. For completeness, we describe the rationale behind these features:

- **Instruction mix features**: collected based on the types and/or frequencies of executed opcodes. We considered four features based on opcodes. Feature INS1 tracks the frequency of opcode occurrence in each of the x86 instruction categories. The top 35 opcodes with the largest difference (delta) in frequency between malware and regular programs were aggregated and used as feature (INS2). Finally, INS3 and INS4 are a binary version of INS1 and INS2 respectively; INS3 tracks the presence of opcodes in each category and INS4 indicating opcode presence for the 35 largest difference opcodes.
- **Memory reference patterns**: collected based on memory addresses used by the program. Feature MEM1 keeps track of the memory reference distance in quantized bins (i.e., creates a histogram of the memory reference distance). The binary version of MEM1 is feature MEM2 that tracks the presence of a load/store in each of the distance bins.
- **Architectural events**: collected based on architectural events. The features collected were: total number of memory reads, memory writes, unaligned memory accesses, immediate branches and taken branches. This feature is called ARCH in the remainder of the paper.

Consistent with the methodology used by earlier works [37, 123], we collected the features once every 10K committed instructions of the running program. The selected frequency (10K) effectively balances complexity and detection accuracy for offline [37] and online [123] detection. Thus, for each program we maintained a sequence of these feature vectors collected every 10K instructions, labeled as either malware or normal.

3.2 Characterizing Performance of Specialized Detectors

In this section, we introduce *specialized detectors*: those that are trained to identify a specific type of malware. First, we investigate whether such detectors' performance exceeds that of *general detectors*, which are trained to classify any type of malware. After establishing that they do indeed outperform general detectors, we proceed by exploring how to use such detectors to improve the overall detection of the system.

We used two different classification algorithms in our experiments: (1) Logistic Regression (LR), which is a simple classification algorithm [60] that separates two classes using a linear boundary in the feature space. The motivation behind using LR is the ease of implementation in hardware; and (2) Neural Networks (NN) which is a network of perceptrons that can be trained to approximate a classification function that is generated from the training data. Note that a single perceptron in NN is equivalent to LR [3]; thus, NN is expected to outperform LR but with the cost of additional implementation complexity. The motivation of using NN is its more effective classificaiton due to its non-linear boundary. Additionally, NN can detect evasive malware when retrained, while LR cannot [79].

The collected feature data for programs and malware is used to train LR and NN detectors. We pick the threshold for the output of the detector, which is used to separate a malware from a regular program, such that it maximizes the sum of the sensitivity (recall) and specificity. Sensitivity is the proportion of malware that the system correctly identifies as malware while specificity is the proportion of regular programs that the system correctly identifies as regular programs [150]. For each detector in this work, we present the threshold values to enable reproduction of our experiments.

Training General Detectors The general detectors are designed to detect any type of malware. Therefore, a general detector is trained using a data set that encompasses all types of malware programs, against another set with regular programs. We trained seven general detectors, one for each of the feature vectors we considered.

Training Specialized Detectors The specialized detectors are designed to detect a specific type of malware relative to the regular programs. We identify the malware type and separate our malware sets into these types based on Microsoft Malware Protection Center classification [109]. The specialized detectors were trained only with malware that matches the detector type, as well as regular programs, so that it would have a better model for detecting the type of malware it is specialized for. For example, the Backdoors detector is trained to classify Backdoors from regular programs only. We chose this approach rather than also attempting to classify malware types from each other because false positives among malware types are not important for our goals. Moreover, types of malware may share features that regular programs do not have and thus classifying them from each other makes classification against regular programs less effective.

3.2.1 Specialized Detectors: Is There an Opportunity?

Intuitively, each malware type has different behaviour allowing specialized detectors to more accurately carry out classification. Thus, in this section, we explore this intuition and quantify the advantage obtained from specializing detectors.

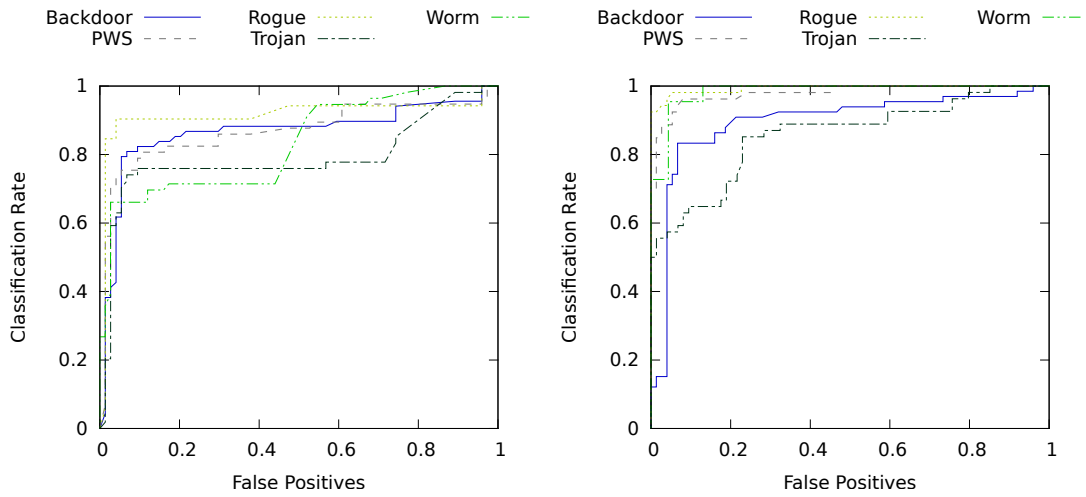
We built specialized detectors for each type of malware we have in the data set (Backdoor, PWS, Rogue, Trojan and Worm). Next, we compared the performance of each of the seven general detectors against each of the specialized detectors performance with respect to classifying the specific malware type for which the specialized detector

was trained. Each comparison between specialized and general detectors uses the same testing set for both of detectors. The testing set includes regular programs and the malware type that the specialized detector was designed for.

We compared the performance of the best performing LR and NN general detector against the best LR and NN specialized detector for each type of malware. Figure 3.2(a) shows the Receiver Operating Characteristic (ROC) curves of the LR INS4 general detector (best performing LR general detector) while Figure 3.2(b) shows the ROC curves for the best LR specialized detectors for each type of malware (MEM1 for Trojans, MEM2 for PWS, INS4 for Rogue, and INS2 for both Backdoor and Worms). The ROC curves represent the classification rate (i.e., Sensitivity) as a function of false positives (100-Specificity) for different threshold values between 0 and 1. In most cases, the specialized detectors outperform the general detector, sometimes significantly.

Figure 3.3 shows the average improvement of the Area Under the Curve (AUC) for each type of malware using the best LR general detector (INS4) and the best NN general detector (INS2). It also shows the performance of the best LR specialized detector, and the best NN specialized detector for each type of malware. Overall, the improvement opportunity is 0.0904 for using specialized LR detectors over general LR detectors and 0.06 for using specialized NN detectors over general NN detectors improving the AUC by more than 9% and 6% respectively. This improvement has a substantial impact on performance. For example, the improvement in Rogue detection, 8% in the AUC, translates to a 4x reduction in overhead needed for detection according to the work metric we define in Chapter 5). In addition, the specialized NN detectors outperform all other detectors.

Figure 3.4 shows the accuracy values for each of the previous AUC when picking the best operating point (maximum sensitivity+specificity). This results also supports



((a)) Best general detector (INS4)

((b)) Best specialized detector

Figure 3.2: ROC improvement opportunity; comparing the ROC of detecting each malware type by the best general and the best specialized detector

the conclusion that specialized NN detectors outperform all other detectors.

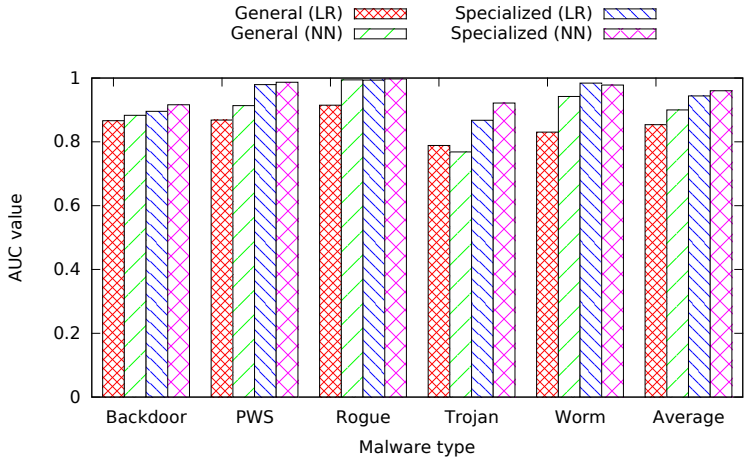


Figure 3.3: AUC improvement opportunity; comparing the AUC of detecting each malware type by the best general and the best specialized detector

Its clear that specialized detectors are more successful than general detectors in classifying malware. However, it is not clear why different features are more successful in detecting different classes of malware, or indeed why classification is at all possible in this low-level feature space. To attempt to answer this question, we examined the weights in the Θ vector of the LR ARCH feature specialized detector for Rogue and

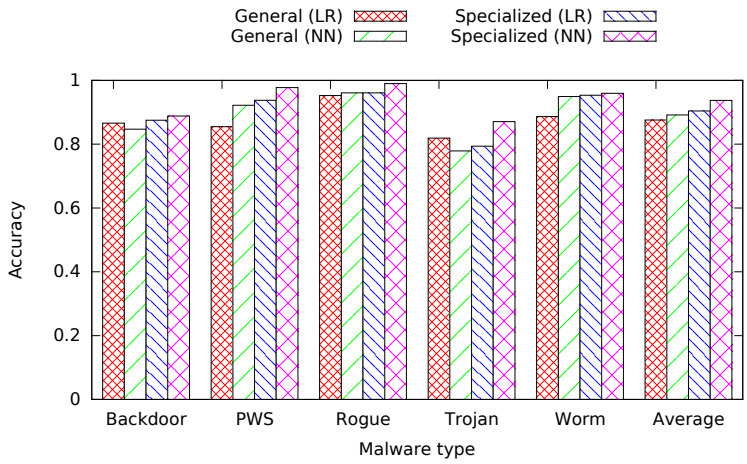


Figure 3.4: Accuracy improvement opportunity; comparing the best accuracy of detecting each malware type by the best general and the best specialized detector

Worm respectively. This feature obtains 0.97 AUC for Rogue but only 0.56 for Worm. We find that the Rogue classifier discovered that the number of branches in Rogue where significantly less than normal programs while the number of misaligned memory addresses were significantly higher. In contrast, Worm weights were very low for all ARCH vector elements, indicating that Worms behaved similar to normal programs in terms of all architectural features.

Chapter 4

Ensemble of Specialized Hardware Mawlare Detectors

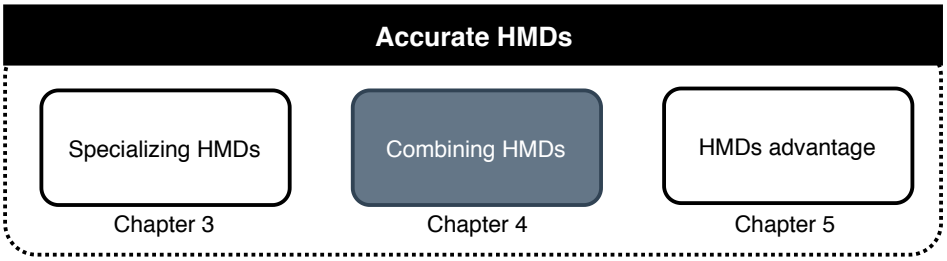


Figure 4.1: Building accurate HMDs project overview

After confirming that specialized detectors perform better than general detectors and identified the features that perform best for each specialized detector (Chapter 3), we investigated combining multiple detectors, general or specialized, to improve the overall performance of the detection to build an ensemble detector. Combining specialized detectors is different from classical ensemble learning where multiple diverse detectors with identical goals are combined to enhance their accuracy. In particular, in our problem, the specialized detectors each answers a different question in the form of: "is the current program a malware of type X?" where X is the type of malware the

detector is specialized for. Combining such detectors requires different forms of combination functions to produce the ensemble decision. We evaluate the performance of the ensemble detectors in both offline and online detection. Furthermore, we analyze the implications on the hardware complexity of the different configurations in Section 4.2.

In summary, in this chapter, we use ensemble learning to improve the performance of the HMD:

- We study using classical ensemble learning by combining multiple general detectors that are trained based on different features.
- We explore combining the specialized detectors for each malware type.
- We investigate a mixed detector by combining general and specialized detectors.

Moreover, we evaluate the hardware complexity of the proposed designs by extending the AO486 open core. We propose and evaluate some hardware optimization to both the Logistic Regression (LR) and Neural Network (NN) implementations.

4.1 Malware Detection Using Ensemble Learning

The next problem we consider is to how to use the specialized detectors to perform overall detection performance. The problem is challenging since we do not know the type of malware (or indeed if a program is malware) during classification. We start with a set of general detectors, each trained using each of our features, and a set of specialized detectors, each trained using one feature and for one malware type. The combining problem considers how to combine the decisions of multiple detectors (base detectors). To combine multiple base detectors, a decision function is used to fuse their result into a final decision. Figure 4.2 illustrates the combined detector components and overall operation.

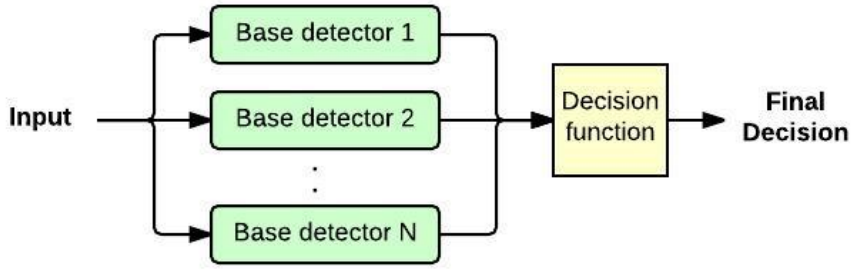


Figure 4.2: Overview of a combined detector; combining the decisions of multiple base detectors using a decision function to produce a final decision

The general technique of combining multiple detectors is called *ensemble learning*; the classic type combines multiple independent detectors that are each trained for the same classification problem (i.e, to classify the same phenomena) [40]. For example, for malware detection, all the general detectors were designed to detect any type of malware. Thus, ensemble learning techniques apply to the problem of combining their decisions directly.

When using specialized detectors, each detector is trained to classify a different phenomena (a different type of malware); they are each answering a different classification question. Given that we do not know if a program contains malware, let alone the malware type, it is not clear how specialized detectors can be used as part of a practical detection solution. In addition, we do not know whether common ensemble learning techniques, which assume detectors that classify the same phenomena, would successfully combine the different specialized detectors.

In order to solve this problem, we evaluate different decision functions to combine the specialized detectors. We focused on combining techniques which use all the detectors independently in parallel to obtain the final output from the decision function. Since all the detectors are running in parallel, this approach speeds up the computation.

4.1.1 Decision Functions

We evaluated the following decision functions.

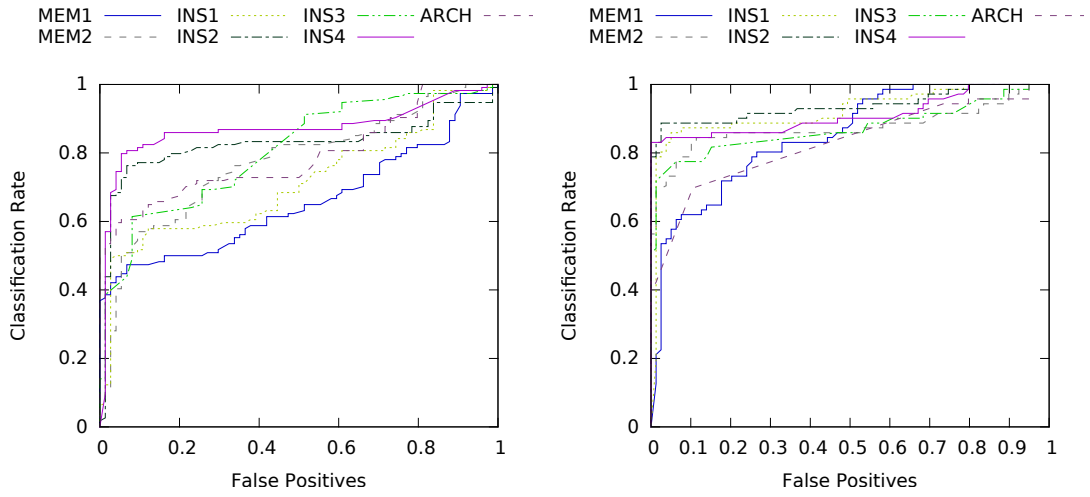
- **Or'ing**: the final decision is malware if any of the base detectors detects the input as malware. This approach is likely to improve sensitivity, but result in a high number of false positives (reduce specificity).
- **High confidence**: the final decision is selected using the Or'ing decision function. However, in this decision function, we select the specialized detector thresholds so that their output will be malware only when they are highly confident that the input is a malware program. Intuitively, specialized detectors are likely to have high confidence only when they encounter the malware type they are trained for.
- **Majority voting**: the final decision is based on the decision of the majority of the base detectors. Thus, if most of them agreed that the program is a malware the final decision will be that it is a malware program.
- **Stacking (Stacked Generalization)**: in this approach, a number of first-level detectors are combined using a second-level detector (*meta-learner*) [170]). The key idea, is to train a second-level detector based on the output of first-level (base) detectors via validation data set. The final decision is the output of the second level detector.

The stacking procedure operates as follows: we from a new data set from collecting the output of each of the base detectors using the validation set. The collected data set consists of every base detector decision for each instance in the validation data set as well as the true classification label (malware or regular program). In this step, it is critical to ensure that the base detectors are formed using a batch of the training data

set that is different from the one used to form the new data set. The second step is to treat the new data set as a new problem, and employ a learning algorithm to solve it.

4.1.2 Ensemble Detectors

To aid with the selection of the base detectors to use within the ensemble detectors, we compare the set of general detectors to each other. Figures 4.3(a) and 4.3(b) show the ROC graph that compares all the LR and NN general detectors respectively. We used a test data set that includes the test sets of all types of malware added to the regular programs test set. The best performing LR general detector uses the INS4 feature vector and the best performing NN general detector uses the INS2 feature vector; we used them as the baseline detectors.



((a)) Logistic regression

((b)) Neural network

Figure 4.3: General detectors comparison; comparing the general detectors performance using ROC curves

We tested different decision functions and applied as input to them different selections of base detectors. An ROC curve based on a validation data set was generated for each base detector to enable identification of the best threshold values for the base detectors. Subsequently, the closest point on the ROC curve to the upper left corner

of the graph, which represents the maximum Sensitivity+Specificity on the training set, was use to select the threshold since we considered the sensitivity and specificity to be equally important. However, for the High confidence decision function, the goal is to minimize the false positives. Therefore, we selected the highest sensitivity value achieving less than 3% false positive rate.

The validation data set used for the general detectors includes all types of malware as well as regular programs. However, for the specialized detector, it only includes the type of malware for which the specialized detector is designed in addition to regular programs. We consider the following combinations of base detectors:

- *General ensemble* detectors: combine only general detectors using classical ensemble learning. General ensemble detectors work best when diverse features are selected. Therefore, we use the best detector from each feature group (INS, MEM, and ARCH), which are INS4, MEM2, and ARCH respectively for LR detectors and INS2, MEM2, and ARCH for NN detectors. Table 4.1 shows the threshold values for the selected LR base detectors which achieve the best detection (highest sum of sensitivity and specificity). The table also shows the threshold values for the selected NN base detectors. Finally, using the same process, we find the best threshold values for the stacking second-level detector to be 0.781 and 0.542 for the LR and NN stacking detectors respectively.

Table 4.1: General ensemble base detectors threshold values. Threshold values, are the values used to classify malware from regular programs based on a probability of a program being a malware; if the probability > threshold: program is classified as malware, else: program is classified as regular program.

	INS	MEM	ARCH
Best Threshold (LR)	0.812	0.599	0.668
Best Threshold (NN)	0.421	0.581	0.622
High Confidence Threshold (LR)	0.893	0.927	0.885

- *Specialized ensemble* detectors: these combine multiple specialized detectors. For each malware type, we used the best specialized detector. Thus, from LR detectors, we selected the detectors trained using MEM1 for Trojans, MEM2 for PWS, INS4 for Rogue, and INS2 for both Backdoor and Worms. On the other hand, for NN detectors, we selected MEM2 for both PWS and Trojans, and INS2 for the other malware types, with the threshold values shown in Table 4.2. In addition, the threshold value for the stacking second-level detector is 0.751 and 0.597 for LR and NN respectively.

- *Mixed ensemble* detector: combines one or more high performing specialized detectors with one general detector. The general detector allows the detection of other malware types unaccounted for by the base specialized detectors. This approach allows us to control the complexity of the ensemble (by limiting the number of specialized detectors) while taking advantage of the best specialized detectors. In our experiments, we used two LR specialized detectors for Worms and Rogue built using the INS4 features vector, because they performed significantly better than the LR general detector for detecting their type. We combine these with an INS4 general detector to build the LR general ensemble detector. For the NN general ensemble detector, we used three specialized detectors for Backdoor, PWS, and Worm built using INS2 features vector along with an INS2 general detector. The

Table 4.2: Specialized ensemble base detectors threshold values. Threshold values, are the values used to classify malware from regular programs based on a probability of a program being a malware; if the probability > threshold: program is classified as malware, else: program is classified as regular program.

	Backdoor	PWS	Rogue	Trojan	Worm
Best Threshold (LR)	0.765	0.777	0.707	0.562	0.818
Best Threshold (NN)	0.473	0.337	0.180	0.760	0.269
High Confidence Threshold (LR)	0.879	0.890	0.886	0.902	0.867

threshold values of the base detectors are shown in Table 4.3 and 4.4 for the LR general ensemble and the NN general ensemble respectively. The threshold value for the stacking second-level detector is 0.5 for LR general ensemble and 0.52 for NN general ensemble.

4.1.3 Offline Detection Effectiveness

As discussed in Section 3.1.2, each program have a feature instance collected for each 10K committed instructions during execution. To evaluate the offline detection performance of a detector, a decision for each instance is first evaluated. As a proof of concept, we consider programs where most of the decisions are malware to be malware; otherwise, the program is considered to be a regular program.

Table 4.5 and 4.6 show the sensitivity, specificity and accuracy for the different LR and NN ensemble detectors using different combining decision functions. It also presents the work and time advantage, which represent the reduction in work and time to achieve the same detection performance as a software detector; we define these metrics more precisely in Section 5.1. The specialized ensemble detector using the stacking decision function outperforms all other detectors built using the same classification method. The LR specialized ensemble achieves 95.8% sensitivity and only 4% false positive rate, which translates to a 24x work advantage and 12.2x time advantage compared to software only detector. On the other hand, the NN specialized ensemble resulted in

Table 4.3: Logistic regression mixed ensemble base detectors threshold values. Threshold values, are the values used to classify malware from regular programs based on a probability of a program being a malware; if the probability > threshold: program is classified as malware, else: program is classified as regular program.

	INS4	Rogue	Worm
Best Threshold	0.812	0.707	0.844
High Confidence Threshold	0.893	0.886	0.884

Table 4.4: Neural networks mixed ensemble base detectors threshold values. Threshold values, are the values used to classify malware from regular programs based on a probability of a program being a malware; if the probability > threshold: program is classified as malware, else: program is classified as regular program.

	INS2	Backdoor	PWS	Worm
Best Threshold	0.421	0.473	0.870	0.269

Table 4.5: Offline detection performance for multiple combined detectors that are trained using Logistic Regression. Performance is represented as sensitivity, specificity, accuracy, work advantage, and time advantage.

	Decision Function	Sensitivity	Specificity	Accuracy	Work Advantage	Time Advantage
Best General	–	82.4%	89.3%	85.1%	7.7	3.5
General Ensemble	Or'ing	99.1%	13.3%	65.0%	1.1	1.1
	High Confidence	80.7%	92.0%	85.1%	10.1	3.7
	Majority Voting	83.3%	92.1%	86.7%	10.5	4.1
	Stacking	80.7%	96.0%	86.8%	20.1	4.3
Specialized Ensemble	Or'ing	100%	5%	51.3%	1.1	1.1
	High Confidence	94.4%	94.7%	94.5%	17.8	9.2
	Stacking	95.8%	96.0%	95.9%	24	12.2
Mixed Ensemble	Or'ing	84.2%	70.6%	78.8%	2.9	2.2
	High Confidence	83.3%	81.3%	82.5%	4.5	2.8
	Stacking	80.7%	96.0%	86.7%	20.2	4.3

Table 4.6: Offline detection performance for multiple combined detectors that are trained using Neural Networks. Performance is represented as sensitivity, specificity, accuracy, work advantage, and time advantage.

	Sensitivity	Specificity	Accuracy	Work Advantage	Time Advantage
Best General	88.7%	88.6%	88.3%	7.8	4.4
General Ensemble (*Stacking)*	83.1%	100%	91.7%	∞	5.9
Specialized Ensemble (*Stacking)*	92.9%	100%	96.5%	∞	14.1
Mixed Ensemble (*Stacking)*	80.2%	98.7%	89.7%	61.6	5.4

92.9% sensitivity and 0% false positive, which translates to unbounded work advantage and 14.1x time advantage compared to software only detector. Thus, NN specialized ensemble have unbounded work advantage, since they have no false positives on our dataset, and 15% faster detection than the LR specialized ensemble.

The Or'ing decision function results in poor specificity for most LR ensembles, since it results in a false positive whenever any detector encounters one. Majority voting was used only for LR general ensembles as it makes no sense to vote when the detectors are voting on different questions. Majority voting performed reasonably well for the LR general ensemble.

For the LR general ensemble detector, Stacking performs the best, slightly improving performance relative to the baseline detector. The majority voting was almost as accurate as stacking but results in more false positives. The mixed ensemble detector did not perform well; with stacking, it was able to significantly improve specificity but at low sensitivity.

4.1.4 Online Detection Effectiveness

Thus far, we have investigated the *offline* detection success: i.e., given the full trace of program execution. In this section, we present a moving window approach to allow real-time classification of the malware following a design in our previous work [123]. In particular, the features are collected for each 10,000 committed instructions, and classified using the detector. We keep track of the decision of the detector using an approximation of Exponential Moving Weighted Average. During a window of time of 32 consecutive decisions, if the decision of the detector reflects malware with an average that crosses a preset threshold, we classify the program as malware.

Online detection performance

Table 4.7: Online detection performance for multiple combined detectors that are trained using Logistic Regression. Performance is represented as sensitivity, specificity, and accuracy.

	Sensitivity	Specificity	Accuracy
Best General	84.2%	86.6%	85.1%
General Ensemble (*Stacking)*	77.1%	94.6%	84.1%
Specialized Ensemble (*Stacking)*	92.9%	92.0%	92.3%
Mixed Ensemble (*Stacking)*	85.5%	90.1%	87.4%

Table 4.8: Online detection performance for multiple combined detectors that are trained using Neural Networks. Performance is represented as sensitivity, specificity, and accuracy.

	Sensitivity	Specificity	Accuracy
Best General	89.2%	85.6%	86.9%
General Ensemble (Stacking)	91.6%	89.9%	90.6%
Specialized Ensemble (Stacking)	93.2%	94.4%	93.8%
Mixed Ensemble (Stacking)	94.4%	89.8%	91.7%

The result of the online detection performance are in Table 4.7 and 4.8. We evaluate candidate detectors in the online detection scenario. The performance for LR detectors is slightly worse for online detection than offline detection, which benefits from the full program execution history. However, for the NN detectors the sensitivity increased, but the specificity and accuracy decreased. The NN specialized ensemble using the stacking decision function outperforms all other detectors by detecting 93.2% of the malware with only 5.6% of false alarms.

Online detection time

Next, we study the time for detecting a malware program in the hardware detector during execution. This time is defined as the number of decision windows (10K committed instructions) that a detector took since a malware started running to classify it as malware. Figure 4.4 shows the cumulative probability distribution of the

detected malware programs as a function of the number of decision windows for both specialized ensemble detectors LR and NN. NN clearly outperforms LR in this metric. For example, NN detects 88% of malware in 500 periods or less, while LR only detects 7%, and over 95% of the malware in 1000 periods or less, while LR detects less than 60%. On average, NN detected malware 5x faster than LR. On a 3 GHz processor, assuming an Instruction Per Cycle (IPC) of 2, this translates to around 500 μ-seconds for NN compared to around 2.5 milliseconds for LR. Thus, NN detectors can alert the software detector more quickly, and limit the opportunity the malware has to do damage to the system.

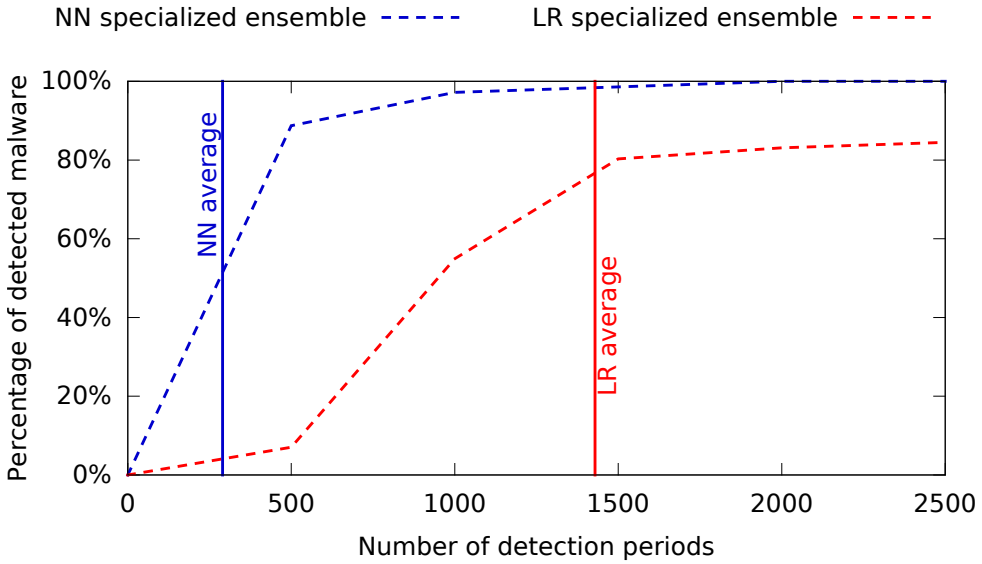


Figure 4.4: Online detection time; time it takes to detect a malwre during it's execution. This Figure shows the cumulative probability distribution of the detected malware programs as a function of the number of decision windows for both specialized ensemble detectors Logistic Regression and Neural Networks.

4.2 Hardware Implementation

We design the ensemble detector using 5 major pipeline stages (shown in Figure 4.5) each of these stages is further pipelined. The first stage of the pipeline (feature

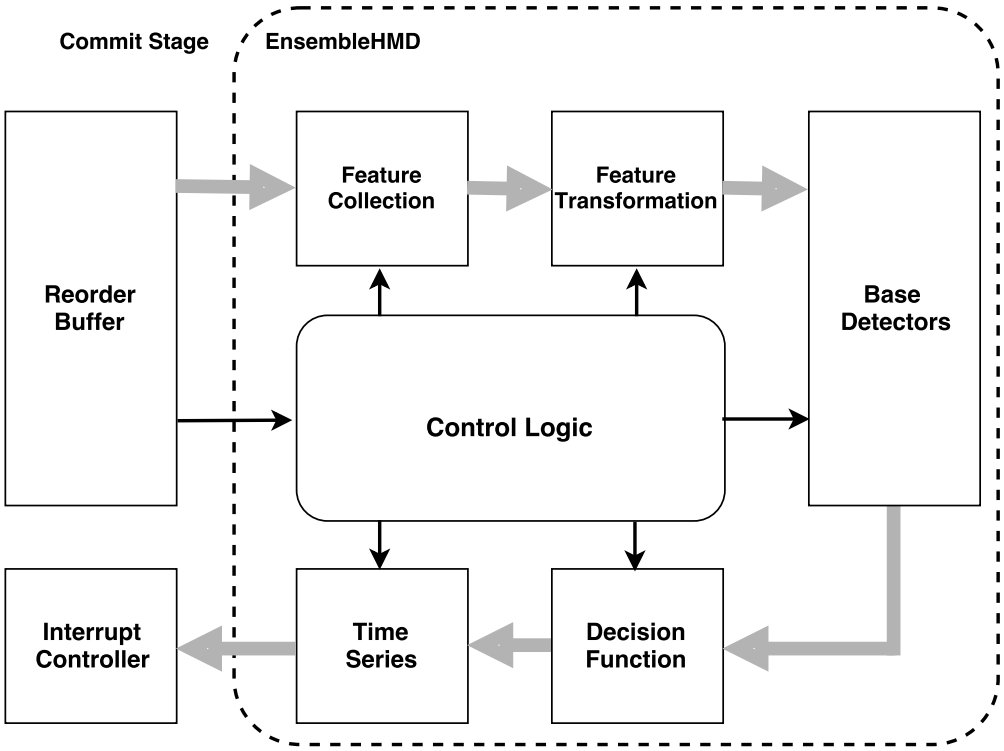


Figure 4.5: Overview of the ensemble malware detection framework attached to the commit stage of the pipeline.

collection) performs feature extraction on the instructions being committed from the tail end of the processor pipeline. For example, the INS4 feature is collected by examining the opcodes of committed instructions and outputting the instruction category to the next stage of the pipeline.

The second stage of the pipeline (feature transform) performs context aware transformation on the raw feature information. For instance, INS2 is generated by taking the INS4 feature and echoing the specific feature if it has not yet been committed during the current detection cycle, transforming it from an integer feature to a binary one. The division of feature collection and feature transform reduces the need for multiple highly similar feature collection units.

The third stage consists of the base detectors (neural networks or perceptrons). Perceptrons require minimal hardware investment [123].This low overhead is achievable

because perceptrons can be optimized into a single accumulator, feature weight look up table and a comparison (Figure 4.6).Typically, perceptrons typical operate using the following formula:

$$\begin{cases} 1 & \text{if } \bar{W} \cdot \bar{X} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\bar{W}$ is the weights vector, $\bar{X}$ is the feature vector, and b is the bias. However, as we commit only 1 feature per cycle we can perform the dot product without actually ever doing a multiply by initializing an accumulator to b then accumulating the weight of each feature when it is committed. Unfortunately, neural networks can not be optimized in the same way. Neural networks require a full multiply accumulate loop and a sigmoid function for each neuron in the network. This causes neural networks to have a significantly higher hardware overhead (as high as 25x for this stage) as compared to perceptrons. Therefore, despite there is an interesting complexity performance trade-off between LR and NN.

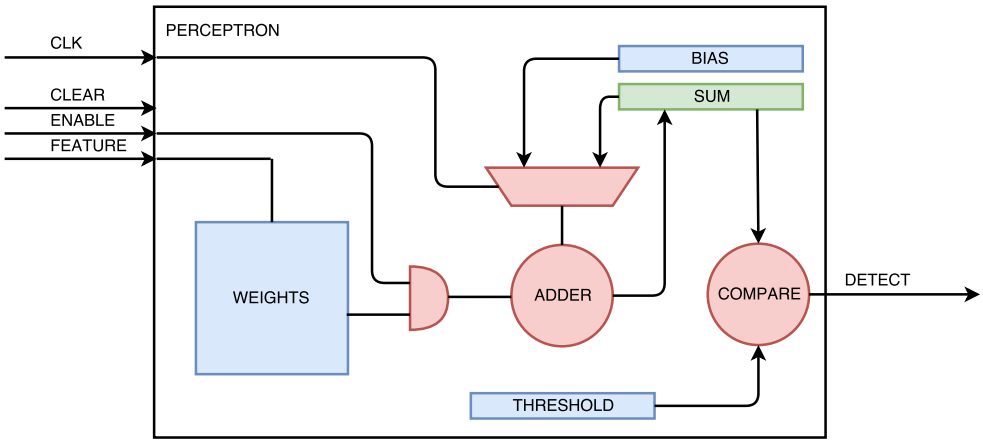


Figure 4.6: Optimized perceptron hardware implementation.

The fourth stage takes the outputs of the base detectors and combines them to form a single prediction. As mentioned in Section 4.1 we explored a number of different

methods for combining the outputs of the base detectors, all of which are simple to implement in hardware. For instance, the most powerful and complex decision function, stacking, can be implemented as a look up table as the number base detectors is small. For example, in our implementation we perform stacking on 5 detectors, using a single 32 bit register to hold all the possible outputs.

Finally, in the fifth stage, time series analysis of the decision function is performed to increase the specificity of the model. In our implementation we use a windowed moving average model. More complex models such as an auto-regressive integrated moving average could possibly provide better results but require a larger hardware budget.

We used the Quartus 2 17.1 software to synthesize implementation of the ensemble detectors attached to the commit stage of the processor pipeline on an DE2-115 FPGA board [1]. Our complete implementation using perceptrons requires a minimal hardware investment. Taking up only 2.88% of logic cells on the core and using only 1.53% of the power compared to an open source processor [4]. Figure 4.7 shows the FPGA layout of the implementation integrated to the processor. The implementation was functionally validated to collect features and classify them correctly. While the detector may be lightweight in terms of physical resources, the implementation required a 9.83% slow down of frequency. However, while this may seem high, the vast majority of this overhead comes from collecting the MEM feature vectors; when we do not collect this feature, the reduction in frequency was under 2%. If feature collection was pipelined over more cycles this cost be significantly reduced or eliminated.

We also note that the area overheads are relative to a small in-order core [4]; compared to a modern core with caches the overhead is likely to be negligible. The frequency overheads are based on the FPGA implementation of the detectors which are known not to correspond directly to delays incurred in a custom implementation of the

logic [95].

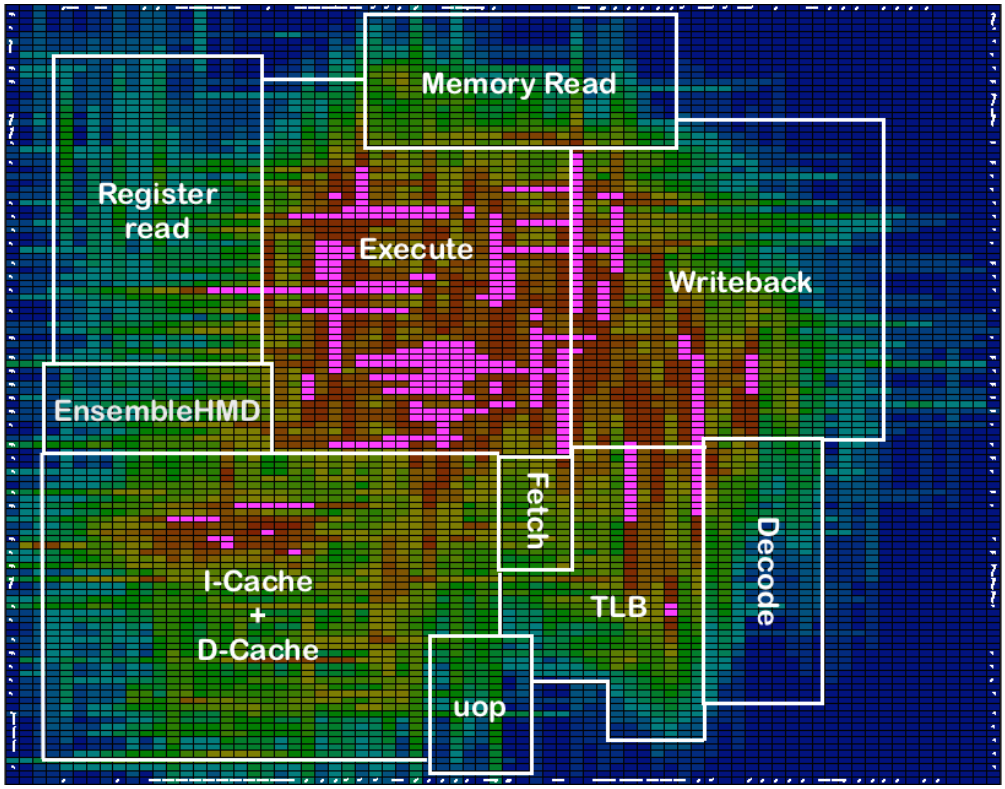


Figure 4.7: FPGA layout of EnsembleHMD integrated into AO486 processor core.

Chapter 5

Advantage of Using Hardware Mawlare Detectors

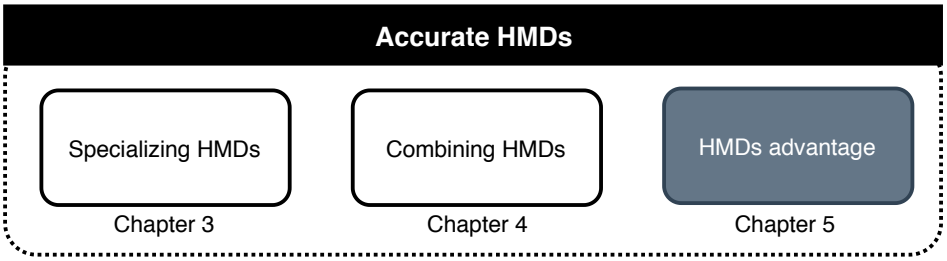


Figure 5.1: Building accurate HMDs project overview

After confirming the advantage of building specialized ensemble detectors in both offline and online detection in terms of accuracy, in this Chapter, we wanted to explore other advantages. In particular, the advantage of ensemble HMDs in a two-level detection system with a more accurate software detector. This advantage can't be directly measured by accuracy: if accuracy were the only metric, we are better off using the software detector alone. The advantage of HMDs result from reducing the overhead necessary for software detection and from prioritizing the efforts of a software detector. To better measure this advantage, we develop metrics that translate detec-

tion performance of HMDs to overhead and time-to-detection advantages of the whole system (Section 5.1). Our ensemble approaches substantially improve the detection of HMDs, reducing the false positives by over half for our best configurations, while also significantly improving the detection rate. As a result, we achieve over 16x reduction in overhead of the two-level detection framework compared to a software only detector. Compared to using a single HMD detector [123, 124], the ensemble detector achieves 2x reduction in overhead and 2.7x reduction in time to detection of the system during online detection.

Furthermore, The paper next conducts a longitudinal study to explore whether detectors trained on a set of malware continue to be effective over time as malware evolves (Section 5.2). We discover that the detection performance degrades substantially, motivating the need for a secure update facility to allow the detector configuration to be updated as malware evolves.

In summary, in this chapter, we define metrics for the two-level detection framework that translate detection performance to expected reduction in overhead, and time to detection. Moreover, we explore the question of whether detectors trained on an old generation of malware would continue to successfully detect malware as it evolves (i.e., from a more recent data set). We discover that the classification performance significantly deteriorates as malware evolves. In addition, we checked if detectors trained on recent malware would be able to detect old malware and the answer was no. These results highlight the need to continuously and securely adapt the learning configuration of the detector to track evolving malware.

5.1 Two-level Framework Performance

We envision our HMD to be used as a first level of a two-level detection (TLD) system. The advantage of this approach is that the second level can clean up false positives that arise due to the fact that the HMD uses low level features and simple classifiers. The hardware detector is always on, identifying processes that are likely to be malware to prioritize the second level. The second level could consist of a more sophisticated semantic detector, or even a protection mechanism, such as a Control Flow Integrity (CFI) monitor [183] or a Software Fault Isolation (SFI) [163] monitor, that prevents a suspicious process from overstepping its boundaries. An interesting possibility is to make the second level detector a cloud based system; if malware is found the results can be propagated to other subscribers in the system. The first level thus serves to prioritize the operation of the second level so that the available resources are directed at processes that are suspicious, rather than applied arbitrarily to all processes.

One possible design point is to use a software only malware detector (i.e., make the second level always on). This detector is likely to be more accurate, but unfortunately, it also incurs high overhead. Thus, the advantage of the HMD is to prioritize the operation of the second level detector to lower its overhead, and reduce the detection time. Improving the detection accuracy of the hardware detector can substantially improve the overall system. Reducing false negatives will lead to the detection of more malware, while reducing false positives will reduce the overhead of the system by avoiding unnecessary invocations of the second level detector.

The performance of an HMD in terms of classification accuracy on its own does not reflect the advantage of using it within a TLD (detection overhead and time to detection). To be able to quantify these advantages, in this section we build a model

of the two level detector and derive approximate metrics to measure its performance advantage relative to a system consisting of software protection only. Our goal is to evaluate how improvements in detection translate to run-time advantages for the detection system. Without loss of generality, we assume that the second level consists of a software detector that can perfectly classify malware from normal programs, but the model can be adapted to consider other scenarios as well.

The first level uses low-level architecture features of the running programs to classify them. This classification may be binary (suspicious or not suspicious) or more continuous, providing a classification confidence value. In this analysis, we assume binary classification: if the hardware detector flags a program to be suspicious it will be added to a priority work list. The software detector scans processes in the high priority list first. A detector providing a suspicion index can provide finer prioritization of the software classifier, further improving the detection advantage.

5.1.1 Assumptions and Basic Models

We call the percentage of positive instances correctly classified malware, Sensitivity (S) of the classifier. Similarly, we call the percentage of correctly classified normal programs the Specificity (C). Conversely, the misclassified malware is referred to as *False Negatives - FN*, while the misclassified normal programs are referredto as *False Positives -FP*. For a classification algorithm to be effective, it is important to have high values of both S and C.

We assume a discrete system where the arrival rate of processes is N with a fraction m of those being malware. We also assume that the effort that the system allocates to the software scanner is sufficient to scan a fraction e of the arriving processes (e ranges from 0 to 1).

In the base case a software detector scans a set of running programs that are equally likely to be malware. Thus, given a detection effort budget e, a corresponding fraction of the arriving programs can be covered. Increasing the detection budget will allow the scanner to evaluate more processes. Since every process has an equal probability of being malware, increasing the effort increases the detection percentage proportionately. Thus, the detection effectiveness (expected fraction of detected malware) is simply e.

5.1.2 Metrics to Assess Relative Performance of TLD

In a TLD, the hardware detector informs the system of suspected malware, which is used to create a priority list consisting of these processes. The size of this suspect list, $s_{suspect}$, as a fraction of the total number of processes is:

$$s_{suspect} = S \cdot m + (1 - C) \cdot (1 - m) \tag{5.1}$$

Intuitively, the suspect list size is the fraction of programs predicted to be malware. It consists of the fraction of malware that were successfully predicted to be malware ($S \cdot m$) and the fraction of normal programs erroneously predicted to be malware $(1-C) \cdot (1-m)$.

Work advantage

Consider a case where the scanning effort e is limited to be no more than the size of the priority list. In this range, the advantage of the TLD can be derived as follows. Lets assume that the effort the system dedicates to detection is $k \cdot s_{suspect}$ where k is some fraction between 0 and 1 inclusive. The expected fraction of detected malware for the baseline case is simply the effort, which is $k \cdot s_{suspect}$ (as discussed in the previous

subsection). In contrast, we know that S of the malware can be expected to be in the $s_{suspect}$ list and the success rate of the TLD is $k \cdot S$. Therefore, the advantage, W_{tld}, in detection rate for the combined detector in this range is the ratio of the detection of the TLD to the software baseline:

$$W_{tld} = \frac{k \cdot S}{k \cdot s_{suspect}} = \frac{S}{S \cdot m + (1 - C) \cdot (1 - m)} \tag{5.2}$$

The advantage of the TLD is that the expected ratio of malware in the suspect list is higher than that in the general process list if the classifier is better than random. It is interesting to note that when S+C=1, the advantage is 1 (i.e., both systems are the same); to get an advantage, S+C must be greater than 1. For example, for small m, if S=C=0.75, the advantage is 3 (the proposed system finds malware with one third of the effort of the baseline). If S=C=0.85 (lower than the range that our features are obtaining), the advantage grows to over 5.

Note that with a perfect hardware predictor (S=1, C=1), the advantage in the limit is $\frac{1}{m}$; thus, the highest advantage is during a period where the system has no malware when m approaches 0. Under such a scenario, the advantage tends to $\frac{S}{1-C}$. However, as m increases, for imperfect detectors, the size of the priority list is affected in two ways: it gets larger because more malware processes are predicted to be malware (true positives), but it also gets smaller, because less processes are normal, and therefore less are erroneously predicted to be malware (false positives). For a scenario with a high level of attack (m tending to 1) there is no advantage to the system as all processes are malware and a priority list, even with perfect detection, does not improve on arbitrary scanning.

Detection success given a finite effort

In this metric, we assume a finite amount of work, and compute the expected fraction of detected malware. Given enough resources to scan a fraction a of arriving processes, we attempt to determine the probability of detecting a particular infection.

We assume a strategy where the baseline detector scans the processes in arbitrary order (as before) while the TLD scans the suspect list first, and then, if there are additional resources, it scans the remaining processes in arbitrary order.

When $e <= s_{suspect}$, analysis similar to that above shows the detection advantage to be $(\frac{S}{s_{suspect}})$. When $e >= s_{suspect}$, then the detection probability can be computed as follows.

$$D_{tld} = S + (1 - S) \cdot \frac{e \cdot N - N \cdot s_{suspect}}{N \cdot (1 - s_{suspect})}. \tag{5.3}$$

The first part of the expression (S) means that if the suspect list is scanned, the probability of detecting a particular infection is S (that it is classified correctly and therefore is in the suspect list). However, if the malware is misclassified (1-S), malware could be detected if it is picked to be scanned given the remaining effort. The expression simplifies to:

$$D_{tld} = S + \frac{(1 - S) \cdot (e - s_{suspect})}{1 - s_{suspect}} \tag{5.4}$$

Note that the advantage in detection can be obtained by dividing D_{tld} by $D_{baseline}$ which is simply e.

Time to Detection

Finally, we derive the expected time to detect a malware given an effort sufficient to scan all programs. Please note that this metric, which quantifies the time for detection in the overall system including the second level detector, differs from the detection time metric introduced in Section 4.1.4 which refers to the time of detection within the hardware detector. In the baseline, the expected value of the time to detect for a given malware is $\frac{1}{2}$ of the scan time. In contrast, with the TLD, the expected detection time is:

$$T_{tld} = S \cdot \frac{s_{suspect}}{2} + (1 - S) \cdot (s_{suspect} + \frac{(1 - s_{suspect})}{2}), \tag{5.5}$$

The first part of the expression accounts for S of the malware which are correctly classified as malware. For these programs, the average detection time is half of the size of the suspect list. The remaining (1-S) malware which are misclassified have a detection time equal to the time to scan the suspect list (since that is scanned first), followed by half the time to scan the remaining processes. Simplifying the equation, we obtain:

$$T_{tld} = S \cdot \frac{s_{suspect}}{2} + (1 - S) \cdot (\frac{(1 + s_{suspect})}{2}), \tag{5.6}$$

Recalling that $T_{baseline} = \frac{1}{2}$, the advantage in detection time, which is the ratio $\frac{T_{tld}}{T_{baseline}}$ is:

$$T_{advantage} = S \cdot s_{suspect} + (1 - S) \cdot (1 + s_{suspect}), \tag{5.7}$$

substituting for $s_{suspect}$ and simplifying, we obtain:

$$T_{advantage} = \frac{1}{1 - (1 - m)(C + S - 1)} \tag{5.8}$$

The advantage again favors the TLD only when the sum of C and S exceeds 1 (the area above the 45 degree line in the ROC graph. Moreover, the advantage is higher when m is small (peace-time) and lower when m grows. When m tends to 0, if $C+S = 1.5$, malware is detected in half the time on average. If the detection is better (say $C+S = 1.8$), malware can be detected 5 times faster on average. We will use these metrics to evaluate the success of the TLD based on the Sensitivity and Specificity derived from the hardware classifiers that we implemented.

5.1.3 Evaluating Two Level Detection Overhead

Next, we use the metrics introduced in this section to analyze the performance and the time-to-detect advantage of the TLD systems based on the different hardware detectors we investigated. We also use them to understand the advantage obtained from increasing the detection accuracy using our ensemble techniques.

The time and work advantages for LR detectors are depicted in Figure 5.2(a) and 5.2(b) as the percentage of malware processes increases. The average time to detect for the LR specialized ensemble detector is 6.6x faster than the software only detector when the fraction of malware programs is low. This advantage drops as more malware is encountered in the system; for example, when 10% of the programs are malware (m=0.1), these advantages drop to 4.2x. We observe that the LR specialized ensemble detector has the best average time-to-detection among LR detectors. The amount of work required for detection is improved by 11x by the LR specialized ensemble detector compared to using the software detector only. Although the LR general ensemble detector had a 14x improvement due to the reduction in the number of false positives, its detection rate is significantly lower than that of the LR specialized ensemble due to its lower sensitivity.

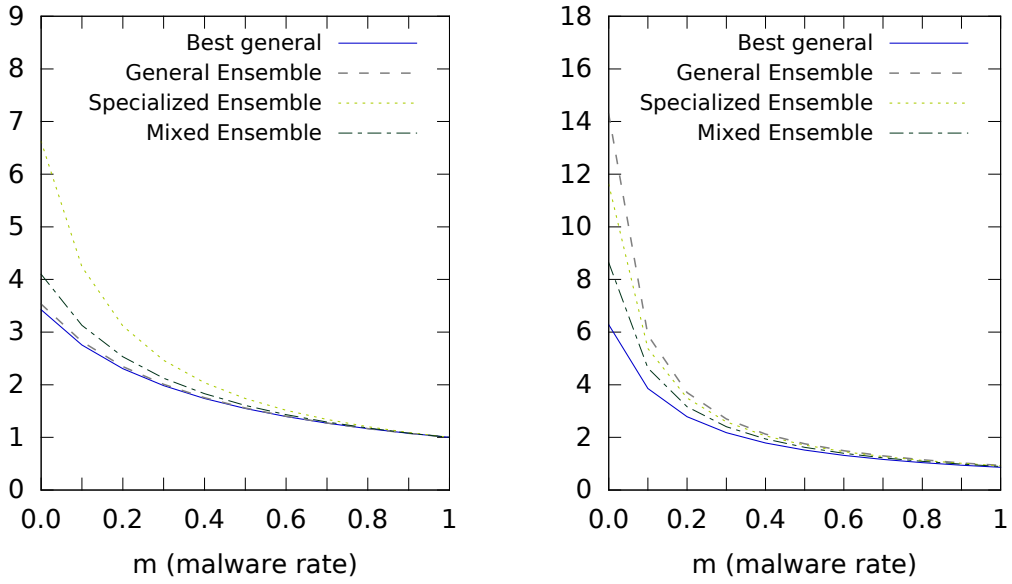
Figures 5.2(c) and 5.2(d) shows the time and work advantage for the NN detectors. The NN specialized ensemble detector outperforms all other detectors with an average time for detection 8x faster than software only detector (1.2x faster than the LR specialized ensemble detector) when the fraction of malware is low. For the same detector, the the work advantage was 16.6x compared to the software detector. This advantage is 1.4x better than the LR specialized ensemble detector.

5.2 Resilience to Malware Evolution

In this section, we study the questions: (1) is a detector trained on a malware training set effective in detecting malware that emerges in the future? If the answer is yes, then there is no need to continue to update the detector to reflect malware evolution. Conversely, if the answer is no, there is a need for a secure mechanism to update the training of the detector over time, even for a hardware-supported detector such as the ones we are studying. (2) When malware evolve, will they insert additional features to the existing ones or they will use different features? These two question were answered for android malwre feature space [5], and we are interested in studying them in the low-level feature space.

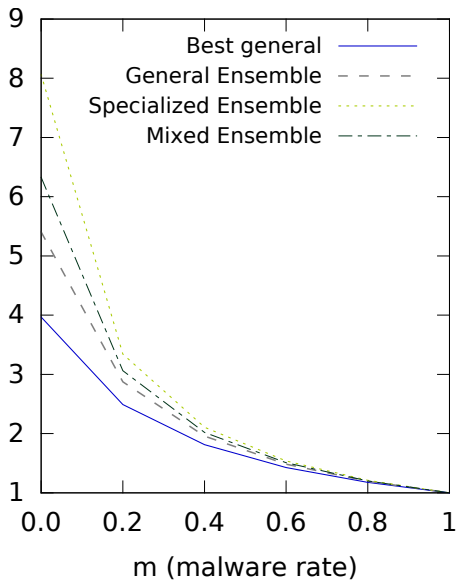
To answer the first question, we use two malware sets with a 4 years difference in the constituent malware. Specifically, we train seven detectors using data sets for malware found in 2009, corresponding to the different feature vectors. We evaluate the malware using two testing data sets: one that contains only malware found in 2009 and another that contains only malware found in 2013.

Figures 5.3(a) and 5.3(b) show the ROC curves using the old and new generations of malware respectively. Examining these figures, it is clear that classification

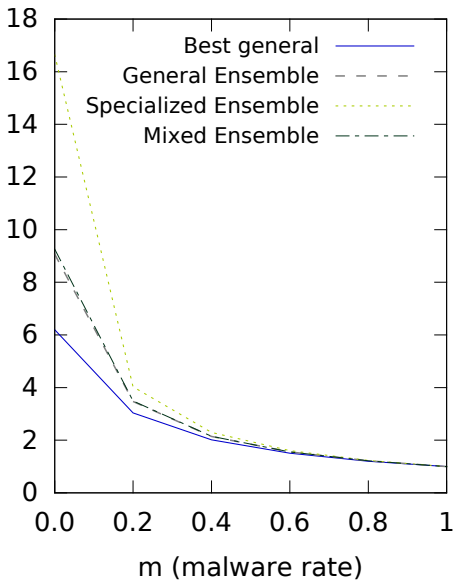


((a)) LR time advantage

((b)) LR work advantage

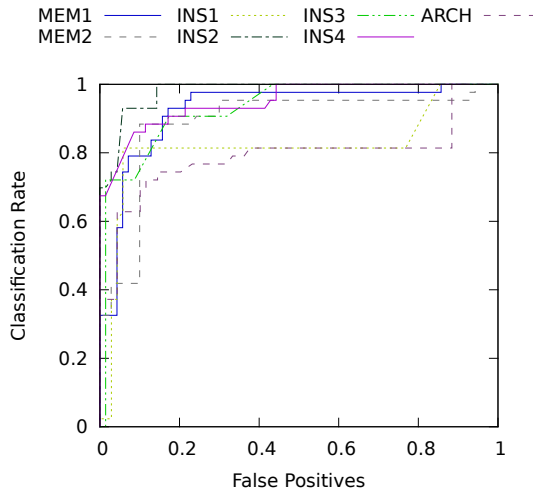


((c)) NN time advantage

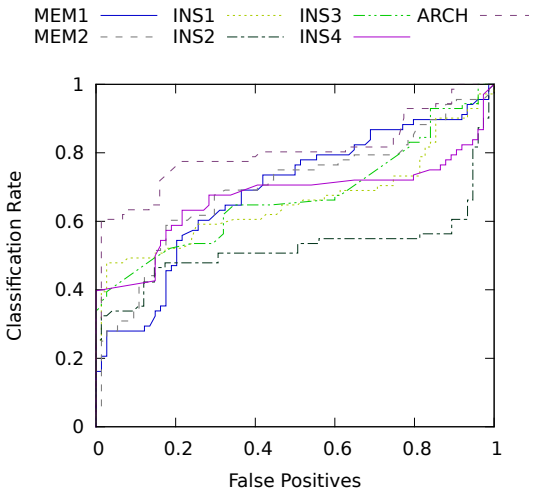


((d)) NN work advantage

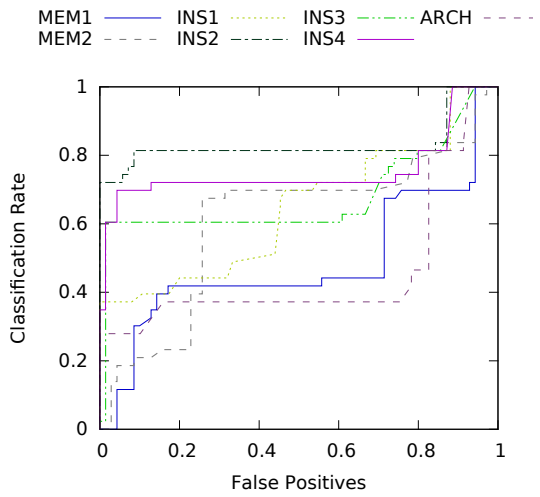
Figure 5.2: Online detection performance, in terms of time and work advantage as a function of malware rate, for different combined detectors and different training algorithms (Logistic Regression and Neural Networks).



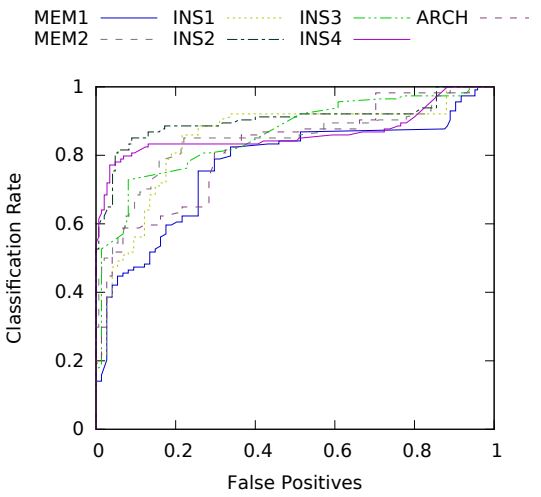
((a)) Old training - old testing data sets



((b)) Old training - recent testing data sets



((c)) Recent training - old testing data sets



((d)) Mixed training - mixed testing data sets

Figure 5.3: Malware evolution; detection performance of detectors trained using old malware in detecting new malware and vice versa.

performance of the detectors trained on the old malware significantly deteriorates when applied to recent malware. Thus, it is essential to develop mechanisms to securely update the hardware detector (e.g., by updating the θ weights vector for the logistic regression) to continue to track the evolution of malware. This secure update can be integrated our system using microcode update function that is used by Intel and other processor manufacturer.

To answer the second question, we used the 2013 malware in training the detectors and the 2009 malware in testing. This detector is used to test if recent malware have representatives features of older malware. The resulting ROC curves is shown in Figure 5.3(c). The results clearly show that using recent malware in training the detector can't detect older malware effectively. Therefore, Figure 5.3(d) show the results when both the 2013 and 2009 datasets are used in both training and testing. The results emphasize the importance of including recent and old malware in the training set of low-level detectors to make sure that they are effective in detecting malware when deployed.

Chapter 6

Reverse Engineering the Hardware Malware Detectors

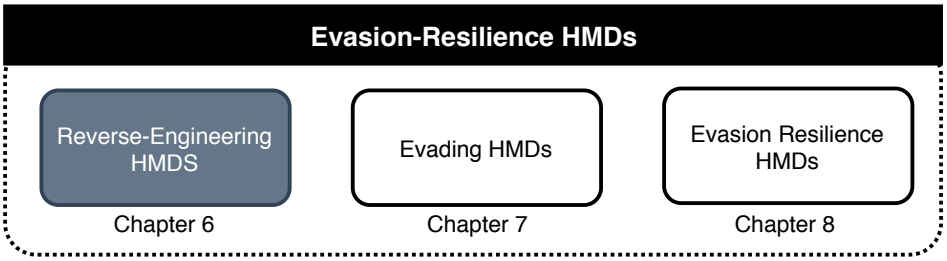


Figure 6.1: Building evasion-resilience HMDs project overview

At a time when malware developers appear to have the upper hand over defenders, as we show in the perivous chapters (Chapters 3, 4, 5) hardware supported malware detection can offer a substantial advantage to defenders because it is always on and has little impact on performance or power [122, 125, 83, 84, 80].

Should HMDs become widely deployed? it is natural to expect that attackers will attempt to evade detection. Therefore, in this chapter and the following chapter (Chapter 7), we explore whether attackers can adapt malware to continue to operate while avoiding detection by HMDs. Intuitively, it should be possible for the attacker

to evade detection if there are no restrictions placed on them, by running a mostly normal program and advancing the attack very slowly. However, we assume that the attacker would like to minimize the impact on the malware execution time since some attacks are time sensitive (e.g., covert and side-channels [30, 63, 129, 75, 47, 115]) or computationally intensive (e.g., Spam or Click Fraud [42, 168]). We describe the threat model and limitations in Section 6.1.

We approach the question of whether malware can evade HMD in the following steps:

1. In this chapter, we will answer the question: Can HMDs be reverse-engineered? Recent results in adversarial classification [162] imply that arbitrarily complex but deterministic classifiers can be reverse-engineered. We confirm that this is the case for HMDs by reverse-engineering a number of detectors under realistic assumptions. We describe our dataset and methodology in Section 6.2, and present and analyze the reverse-engineered detectors in Section 6.3.

2. In Chapter 7, having a model of the detector, can malware developers modify malware to avoid detection? Evading the detection by changing the behavior of the malware is known as mimicry attacks [21, 165]. We show (in Section 7.1) that existing HMDs can be rendered ineffective using simple modifications to the malware binary.

3. Finally, in Chapter 8, after showing that the current generation of HMDs is vulnerable to evasion, we explore whether new HMDs can be constructed that are robust to evasion. In particular, we propose in Section 8.1 a new resilient HMD (RHMD) organization that uses multiple diverse detectors and switches between them unpredictably. We show that RHMDs that use even simple base detectors

are resilient to both reverse-engineering and evasion. Furthermore, this resilience increases with the number and diversity of the base detectors.

The problem of evasive malware detection has been considered in the context of software malware detectors [165, 21, 149]. Moreover, some existing HMD proposals discuss the possibility of malware evasion [38, 76]. However, ours is the first study that explores this important question regarding HMDs in detail and develops solutions to it. We note that while our experiments target HMDs, the underlying evasion problem exists in the context of any adversarial classification problem [162]. Our work advances the state of the art in general, not just for HMDs: we show systematically that reverse-engineering is possible (Chapter 6), we develop techniques that use the result of reverse-engineered detectors to efficiently evade detection (Chapter 7), and we introduce evade-retrain games and study their resilience to evasion.

In summary, in this chapter, we show that it is possible to accurately reverse engineer HMDs, regardless of their complexity.

6.1 Threat Model and Limitations

We assume an adversarial attack model which starts with the adversary attempting to reverse engineer the classifier (figure out the decision boundaries of the model that is used for detection). We assume that the attacker has access to a machine with a similar detector as the victim machine. This allows the attacker to observe the behavior of the classifier for given programs (whether malware or normal programs). With a model of the detector, the attacker can attempt to generate evading malware that hide themselves by changing some of their characteristics (feature values). Such evading mechanism is known as *mimicry* attacks [21, 165], which can be in the form

of no-op insertion, code obfuscation by the attackers, or calling benign functions in the middle of the malicious payload [74].

We assume that the attacker that undertakes malware rewriting as part of a mimicry attack is interested in maintaining reasonable performance of the malware. If this assumption is not true, an attacker can simply run a normal program with embedded malware, that advances the malware program arbitrarily slow (e.g., 1 malware instruction every N normal instructions where N is arbitrarily large), making detection impossible. Note that this is a limitation of all anomaly detectors, and not only HMDs. This assumption is also reasonable for important segments of malware such as: malware that is time sensitive (e.g., that performs covert or side-channel attacks [129, 75, 47, 115]) and malware that is computationally intensive such as that executing on botnets being monetized under a pay-per-install model [22] (e.g., Spam bots or Click fraud). Such malware have a utility to the malware writer proportional to their performance.

6.2 Data and Methodology

We collected samples of malware (from the MalwareDB malware set [118]) and normal programs to use in our study. The downloaded malware data set consisted of 3000 malware programs. For regular program samples, we used Windows programs since the malware programs that we use are Windows-based. The regular program set contains a variety of applications including browsers, text editing tools, system programs, SPEC 2006 benchmarks [56], and other popular applications such as Acrobat Reader, Notepad++, and Winrar. In total, the non-malware data set contains 554 programs. Both malware and regular programs data sets were divided into 60% *victim training*, 20% *attacker training* for the reverse engineered detector, and 20% *attacker testing* of

the detector. To ensure that there is no bias in the distribution of malware programs across the sets, each set includes a randomly selected subset of malware samples from each type of malware in the overall data set.

The data was collected by running both malware and regular programs on a virtual machine with a Windows 7 operating system. To allow malware programs to carry out their intended functionality, the Windows security services and firewall were disabled. Furthermore, the dynamic trace of executed programs was collected using Pin instrumentation tool [34]. Unlike mobile malware where many malware samples require user interaction and necessitate special efforts to ensure correct behavior [77], we observed that the vast majority of our windows/desktop malware operated correctly (through manual inspection and examination malware behavior during run-time); several malware samples tripped the intrusion detection monitoring systems on our network as they attempted to discover and attack other machines, until we separated the environment into an independent subnet.

The collected trace duration for each executed program was 5000 system calls or 15 million of committed instructions, starting after a warm-up period, whichever is reached first. While ideally, we would have liked to run each program longer, we are limited by the computational overhead; since we are collecting run-time behavior of the programs using dynamic profiling information through Pin within a virtual machine, collection requires several weeks of execution on a small cluster and produces several terabytes of compressed profiling traces. Training and testing are also extremely computationally intensive. We believe that this data set is sufficiently large to establish the feasibility and provide trustworthy experimental results.

We collected different feature vectors, specifically:

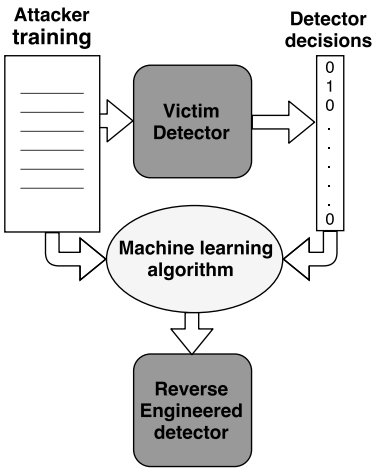
- Executed instruction mixes (called *Instructions* in the rest of the paper): this feature tracks the frequency of instructions that show the most different frequency (delta) between normal programs and malware in the training set;
- Memory address patterns (called *Memory* in the rest of the paper): this feature tracks the distribution of memory references organized in bins based on the address difference between consecutive memory accesses; and
- Architectural events (called *Architectural* in the rest of the paper): tracks the numbers of different architectural events occurring in an execution period such as unaligned memory accesses, and taken branches.

These features are modeled after those used in prior HMD studies [38, 122].

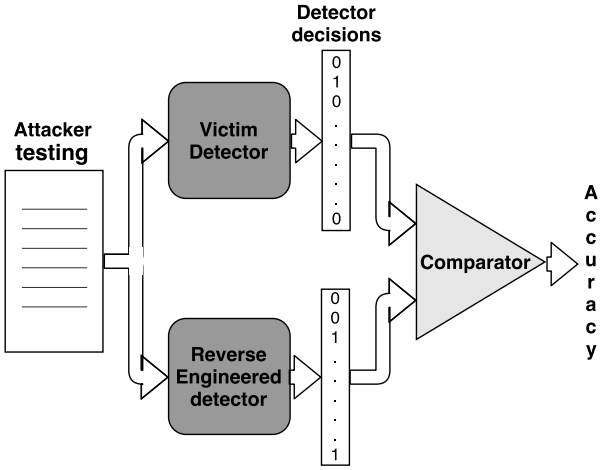
6.3 Reverse-Engineering HMDs

In this section, we demonstrate that we can successfully reverse-engineer HMDs based on supervised learning (e.g., similar to those presented by Demme et al. [38], Ozsoy et al. [122], and some of the detectors used by Kazdagli et al. [77]). Reverse-engineering the malware detector allows the adversary to construct a model of the HMD. The model is necessary to be able to methodically develop evasive malware. We assume that the adversary has the ability to query the targeted detector; if they do not, the problem becomes NP-Hard [162].

Figure 6.2(a) shows the steps in reverse-engineering a detector. First, the adversary prepares a training data set that is composed of both regular and malware programs: this is a separate data set from the one used to train the classifier, which is unknown to the attacker. Next, the adversary uses this data set to query the victim



((a)) Reverse-engineering of victim



((b)) Evaluating reverse-engineered detector

Figure 6.2: Overview of reverse-engineering the HMDs process as well as the process of evaluating the reverse-engineered detector.

detector and records the victim's detection decisions. The decisions are used as the label for the data as we construct the reverse-engineered detector. Finally, the adversary may use different machine learning classification algorithms trained with the labeled data to build the new reverse-engineered detector.

Figure 6.2(b) shows the evaluation of the reverse-engineered detector. The adversary first prepares an attacker testing data set as described above. Next, both the original detector and reverse-engineered detector are queried using the attacker testing

data set. Finally, the percentage of equivalent decisions made by the two detectors is calculated. Note that from the adversary point of view, it does not matter if the detector is classifying malware and regular programs correctly; rather, the attacker desires to mimic the classification of the victim detector and it evaluates success on that basis.

For most of our studies, we evaluate baseline detectors that use logistic regression (LR) and neural networks (NN); the methodology naturally generalizes to other classification algorithms. We implemented the NN classifier as a multi-layer perceptron (MLP) with a single hidden layer that has a number of neurons equal to the number of features in the feature vector. We use the *tanh* function as the activation function. The rationale for selecting these two algorithms is that prior studies showed that LR performs well and has low complexity, facilitating hardware implementations [122]. NN features more complex classification ability, capable of producing a non-linear classification boundary. These detectors allow us to contrast the impact of the detector complexity on both reverse-engineering process and mimicry attacks. For some studies, we use other classifiers to illustrate some generalizations of our conclusions.

The victim data set is used to train different detector instances using each of the two algorithms for each of the three different features, resulting in six detectors. The detectors take a feature vector as an input in order to produce a 0 or 1 decision indicating whether the program is a regular or malware program respectively.

Figure 6.3 shows the performance of the detectors in classifying malware and regular programs using area under the curve (AUC) and the accuracy of the classification metrics for each of the detectors. Note that this figure shows the performance of the baseline HMD which we will be attempting to reverse-engineer. AUC is the area under the Receiver Operating Characteristics (ROC) curve which plots the sensitivity of the detector against the percentage of false positives; the larger the AUC, the better

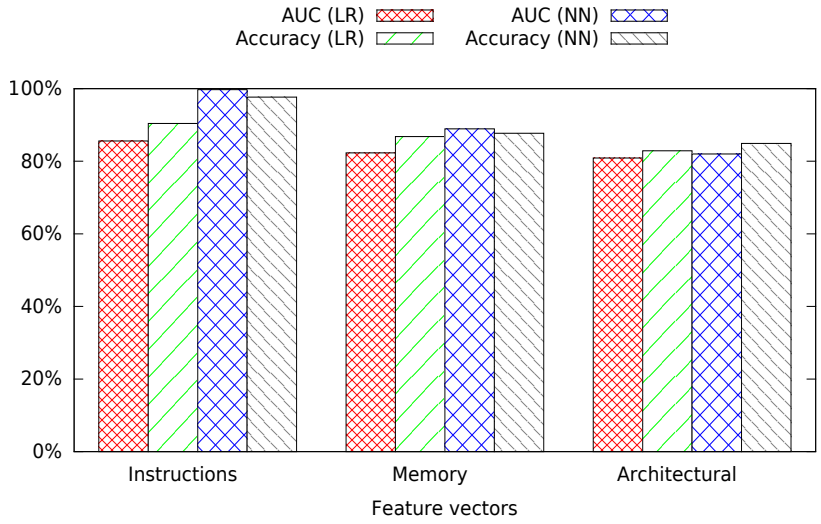


Figure 6.3: Performance of individual detectors (general detectors) for different feature vectors. Performance is represented using AUC values and Accuracy.

the classification. Accuracy refers to the point on the ROC which maximizes the accuracy (percentage of decisions that are made correctly). It is a more direct measure of performance since the HMD classification threshold will be typically set to perform at or near this optimal point.

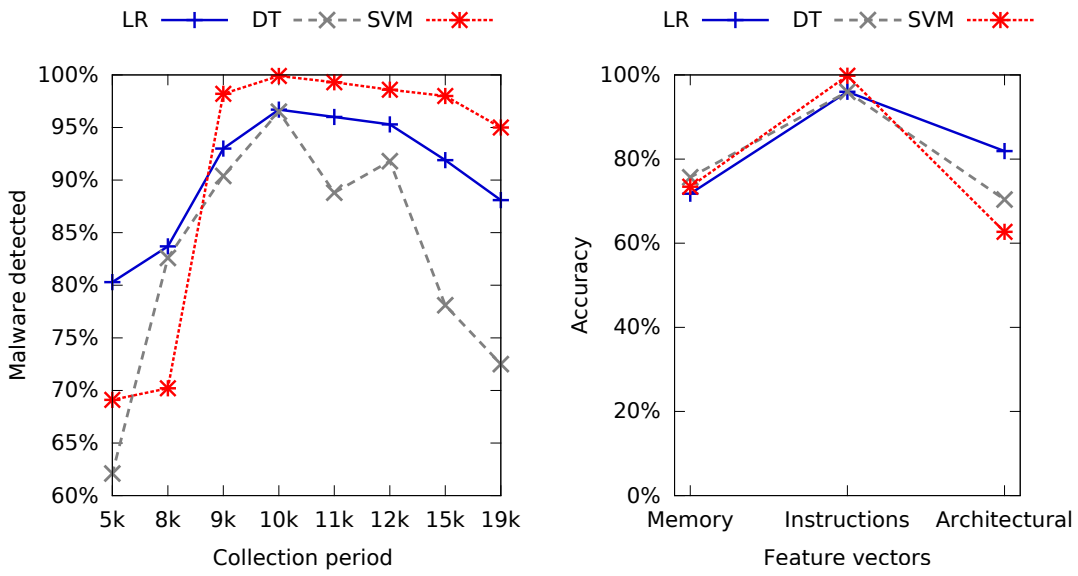
We assume that the attacker does not know the details of how the target victim detector was trained. Thus, they do not know important configuration parameters of the detector including: (1) the size of the instruction window that is used to collect the features; the detector collects the feature over a collection window, typically measured in thousands of instructions, and then uses these features for classification; (2) the specific feature used for the classification. However, we assume that the attacker has a set of candidate features that includes the feature used by the target detector; and (3) the classification algorithm used by the target detector. Importantly, the attacker has access to a machine with a similar detector so they can test hypotheses and evaluate the success of the mimicry attacks. Next, we show how the attacker can reverse-engineer the detection period and the features used in training the target detector.

6.3.1 Target Detector Classification Period

The classification period refers to the size of the instruction window used to collect the classification features. Prior work [38] has shown that a classification period of about 10K instructions works well for supervised learning classifiers, but a detector may be trained with a different classification period. For this experiment, we used a classifier built using the Instruction mix feature which we assume the attacker knows (later we relax this assumption). The target detector collection period is 10K. We prepare multiple pairs of attacker testing and training datasets, using different collection periods. Next, we train a reverse-engineered detector using different data sets and evaluate its accuracy. For each of the attacker data sets, we construct three reverse engineered detectors using three different machine learning algorithms, which are: LR, decision tree (DT), and support vector machine (SVM). The results of this experiments are shown in Figure 6.4(a). The results show that the highest accuracy for reverse-engineering for each of the machine learning algorithms used is when the collection period is the same as the victim's collection period (10K). Thus, by using an experiment such as this one, the attacker can infer the victim's collection period.

6.3.2 Target Detector Feature

Malware detectors can be built using different features. In this subsection, we explore the possibility of reverse-engineering the feature vector used by the victim detector only by querying the victim detector. We use a detector based on the Instruction mix feature with a classification window of 10K instructions. We prepared multiple pairs of attacker testing and training data sets using the same collection period (10K), but using different feature vectors. Next, we construct reverse-engineered detectors us-



((a)) Collection periods

((b)) Features

Figure 6.4: Reverse-engineer victim detector configurations: (a) collection period and (b) feature vector

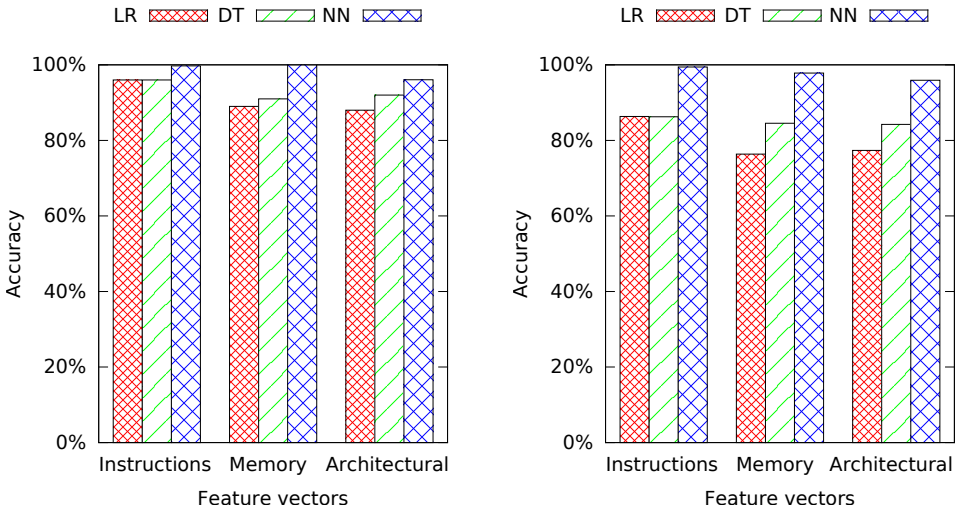
ing the attacker training data sets labeled with the output from the victim detector as malware or regular program. For each of the attacker data sets, we constructed three detectors using different machine learning algorithms, which are: LR, Decision Tree (DT), and Support Vector Machine (SVM). The results of this experiment are shown in Figure 6.4(b). The results show that the highest accuracy is achieved when the feature vector is the same as the victim's feature vector (Instructions). We conclude that the victim HMD features can be successfully reverse-engineered.

Note that at the correct value of the feature and period, it is possible to obtain 0-error reverse-engineering in our experiments. This is consistent with results from PAC learning theory which we overview in Section 8.2. Although we showed how to separately reverse-engineer the classification period (assuming that the classification feature is known) and the classification feature (assuming the classification period is known), we can also jointly reverse-engineer them both. The process involves constructing detectors

with different classification features and periods, and finding the detector that maximizes the reverse-engineering accuracy.

6.3.3 Performance of Reverse-engineered HMD

In the next set of experiments, we evaluate the performance of the reverse-engineered detectors. We reverse-engineer LR and NN detectors, but the reverse-engineered detector is constructed using three different machine learning algorithms, which are LR, Decision Trees (DT), and NN. The results for reverse-engineering of the LR and NN detectors are shown in Figure 6.5(a) and Figure 6.5(b) respectively. The results show that NN can reverse-engineer both types of detectors with high accuracy (e.g., less than 1% error for all features for LR). The performance is somewhat lower for NN since the separation criteria used in the classification is more complex and therefore more difficult to reverse-engineer accurately.As can be expected, the simpler linear detector (LR) cannot effectively capture the non-linear behavior of NN, consistent with PAC learning theory as we discuss in Section 8.2.



((a)) Logistic regression

((b)) Neural networks

Figure 6.5: Reverse-engineering efficiency; accuracy of reverse-engineering the victim detectors using different machine learning algorithms.

Chapter 7

Evading Hardware Malware Detectors

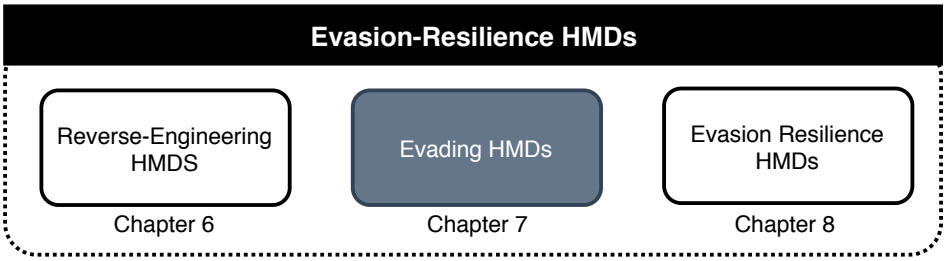


Figure 7.1: Building evasion-resilience HMDs project overview

After confirming that HMDs can be reverse-engineered effectively, in this chapter we will explore creating evasive samples of malware using the reverse engineering information. In particular, having a model of the detector, can malware developers modify malware to avoid detection? Evading the detection by changing the behavior of the malware is known as mimicry attacks [21, 165]. We show (in Section 7.1) that existing HMDs can be rendered ineffective using simple modifications to the malware binary.

As a first step to find a solution for the evasion problem, we have tried to retrain the HMDs using evasive samples to see if they can generalize to detect evasive malware. Therefore, we tried to answer the question: Can the malware evade detection even if the detector is retrained with some samples of the evasive malware? We show in Section 7.2 that for simple evasion strategies that can fool a given detector, retraining a logistic regression (LR) detector does not result in effective classification of evasive malware, unless the detection performance on normal malware is sacrificed. On the other hand, more sophisticated detectors such as Neural Networks (NN) can be successfully retrained, but the attacker is still able to reverse-engineer the retrained detector and evade it again.

In summary, in this chapter, we show that once an HMD has been reverse engineered, malware can effectively evade it using low overhead evasion strategies. This result brings into question the effectiveness of existing HMDs. Furthermore, we show that simple linear HMDs such as LR cannot be retrained to adapt to evasive malware. More complex classifiers such as NN can adapt better but may break down after several generations of evasion and retrain. Moreover, new malware can still reverse-engineer and evade even such classifiers.

7.1 Developing Evasive Malware

After reverse-engineering the victim detectors, the next step that attackers are likely to take is to develop systematic transformations of their malware that can evade detection by these detectors. The malware developers may modify their malware in any way, to attempt to produce behavior in the feature space of the detector that causes them to be classified as normal. Possible strategies to accomplish this goal include using

polymorphism to produce different binaries [179], or adding instructions that do not affect the malware state.

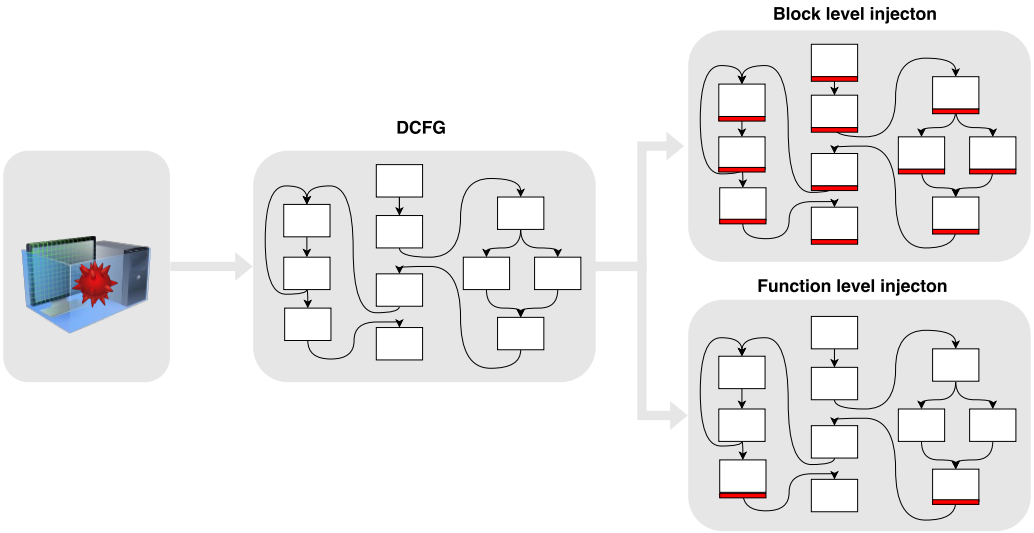


Figure 7.2: Methodology for generating evasive malware (an overview).

Since we are working with actual malware binaries, we do not have the source available to apply general transformations. Moreover, most of the malware is obfuscated making decompilation difficult and challenge binary rewriting tools. To address these challenges, we developed a methodology to dynamically insert instructions into the malware execution in a controllable way (Figure 7.2). In particular, we construct the Dynamic Control Flow Graph (DCFG) of the malware by instrumenting it through the PIN tool [34]. Next, we add instructions into the control flow graph in a way that does not affect the execution state of the program.

The injected instructions must change feature vector in a controlled way based on the reversed-engineered classifier to attempt to move the malware across the classification decision boundary to be classified as normal. For the Instruction feature, injection of opcodes increases the weight of the corresponding feature directly. For the memory feature, insertion of load and store instructions with controlled distances changes the histogram of memory reference frequencies. For architectural features, the effects may

not be directly controllable. For example, increasing the cache hit rate or the branch predictor success rate requires inserting code segments that will generate cache misses or predictable branches respectively. Without loss of generality, all of our experiments use the instruction feature. When attempting to change multiple features, the evasion code must combine these strategies (for example, alternating their use at different injection points).

We explored two approaches for inserting the instructions: (1) Block level: insert instructions before every control flow altering instruction. Note that the instructions inserted at that point in the program are executed every time that control flow instruction is reached (i.e., we do not change the instructions that are injected at a particular point in the program, once they are injected); and (2) Function level: we insert instructions before every return instruction.

Random instruction injection: Although we do not expect this strategy to succeed, we first check if injecting randomly chosen instructions in the malware programs is sufficient to evade detection to establish that the injection must be specific to the detector. Each malware program data set is divided into two sets based on whether the victim detector successfully detected them without modification. Each of the data sets is modified using our framework to inject the additional instructions and retested; the results are shown in Figure 7.3. Clearly, injecting random instructions at the basic block level or at the function call level does not help in evading detection.

Reverse-engineering driven instruction injection: The next set of strategies we use is to exploit the information in the structure of the reverse-engineered detector to attempt to avoid detection. Ideally, we would like to make the malware appear close to regular programs in terms of behavior so that the detector cannot successfully identify it. Based on the parameters of the reverse-engineered detector (from Section 6.3), we

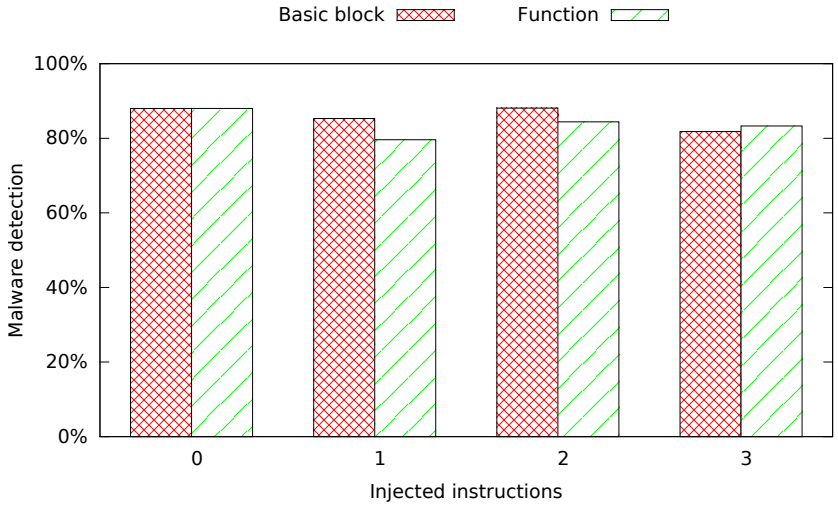


Figure 7.3: Detection performance of evasive malware created using random instruction injection

form a strategy for which instructions we can inject to make the detection difficult.

To understand the rationale behind the instruction selection, we must explain some details about the operation of LR. LR is defined by a vector θ that identifies the linear separation between the points being classified. The weights of the elements of the vector determine the relative importance of these elements. Since we are only adding instructions, we pick the instructions whose weights are negative to move the malware towards the other side of the separation line. In this first strategy, we inject only the instruction with the least weight in the vector.

Figure 7.5(a), shows the percentage of malware detected by both the original and reverse-engineered detectors, after injecting the malware using the information for the reverse engineered detector at the basic block and at the function call levels. We observe that the modified malware evades detection by both detectors.

We conduct a similar experiment for the NN detectors, where the classifier is not clearly defined by a single vector and the separation plane is not linear. We develop a heuristic approach to identify candidate instructions for insertion. Figure 7.4 shows a

NN with one hidden layer. Each circle in the Figure represents a neuron in the network; input, hidden, and output neurons. To compute the overall weight contributed by a single input we multiple out its contributions to the eventual output of the network and sum out these products. For the example in Figure 7.4, the weight of input I1 can be estimated as:

$$w_1 = w_{11}^1 \times w_1^{out} + w_{12}^1 \times w_2^{out} + w_{13}^1 \times w_3^{out}$$

More generally, for input j, the weight is:

$$w_j = \Sigma_{i=1}^{n} w_{ji} \times w_i^{out}$$

With multiple hidden layers, we must add all the factors on all the paths to which a given input contributes.

The procedure above allows us to collapse the description of the NN into a single vector that summarizes the contributions of each feature. This allows us to use the same strategies for instruction selection that we used in LR; for example, we can select the instruction with the most negative weight for insertion. However, for NN this is an approximate strategy; for LR, if we inject more of the negative weight instructions we are guaranteed to monotonically decrease as the dot product of θ and the collected feature from the malware execution becomes increasingly more negative. However, since the separation surface of NN is non-linear, the same cannot be guaranteed.

Figure 7.5(b) shows the percentage of modified malware detected by the NN victim and reverse-engineered detectors. While the evasive strategy also works in this case, it is slightly less effective; with 2 injected instructions per basic block, we can evade detection 80% of the time.

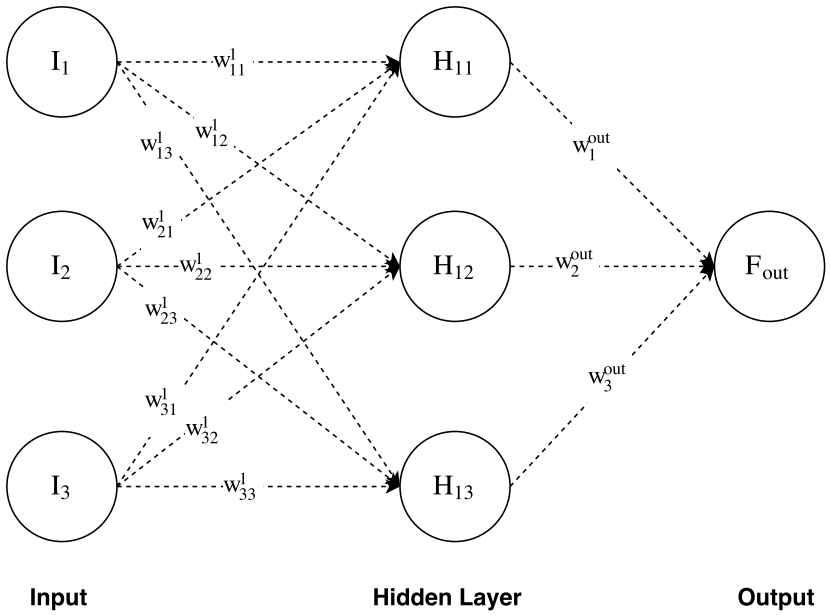
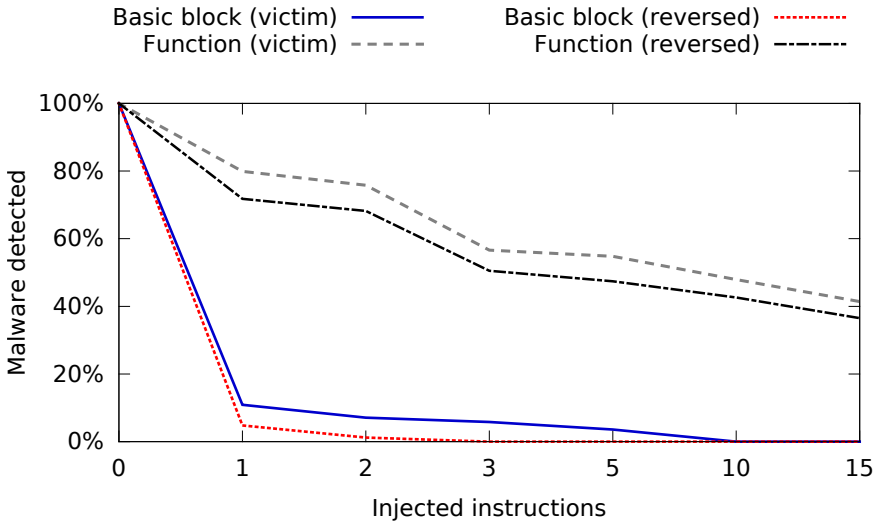


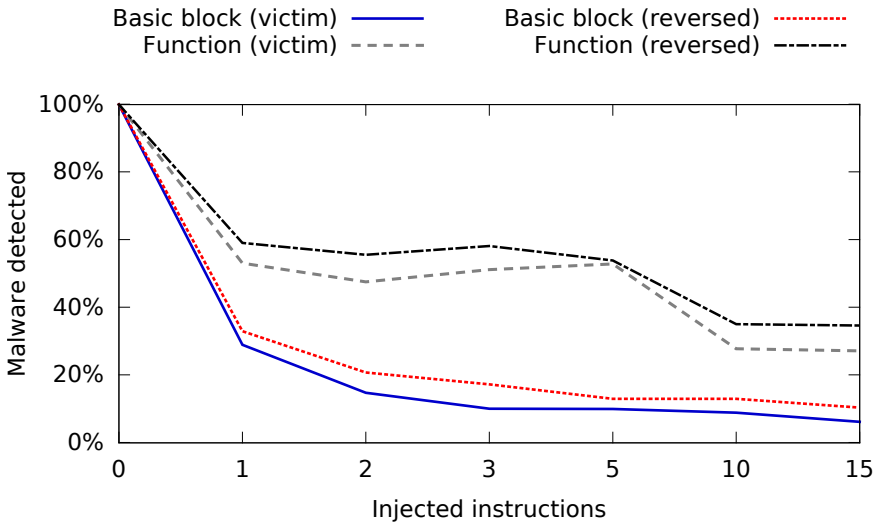
Figure 7.4: Neural network with one hidden layer overview

As we explained in our threat model (Section 6.1), we assume that the attacker is interested in maintaining the performance of the malware, and does not want to arbitrarily slow it down to evade detection. Figure 7.6, shows the static and dynamic overhead of injecting instructions both at the basic block level and the function call level. Inserting a single instruction at the basic block level was effective in evading detection for most malware for LR; both the static overhead (increase in the text segment of the executable) and the dynamic overhead (increase in execution time) are about 10%.

We also consider selecting the instruction for injection among all the instructions with a negative weight, with a probability proportional to the weight; we call this strategy the *weighted injection strategy*. Figure 7.7, shows the percentage of malware detected by the victim, after weighted injection of the malware using the information for both the reverse-engineered detector and the victim detector at the basic block and at the function call levels. The evasion success using the reverse-engineered detector is almost equal to the success when using the actual victim detector. The advantage of



((a)) Logistic regression



((b)) Neural networks

Figure 7.5: Detection performance of evasive malware using the least weight instruction injection method

this strategy is that it makes it more difficult to detect the evasion if the detector is retrained as explained in the next section.

7.2 Retraining Victim Detectors

The results of the previous section demonstrate that existing HMDs that use supervised learning [38, 122] can be fairly easily evaded. The next question we consider

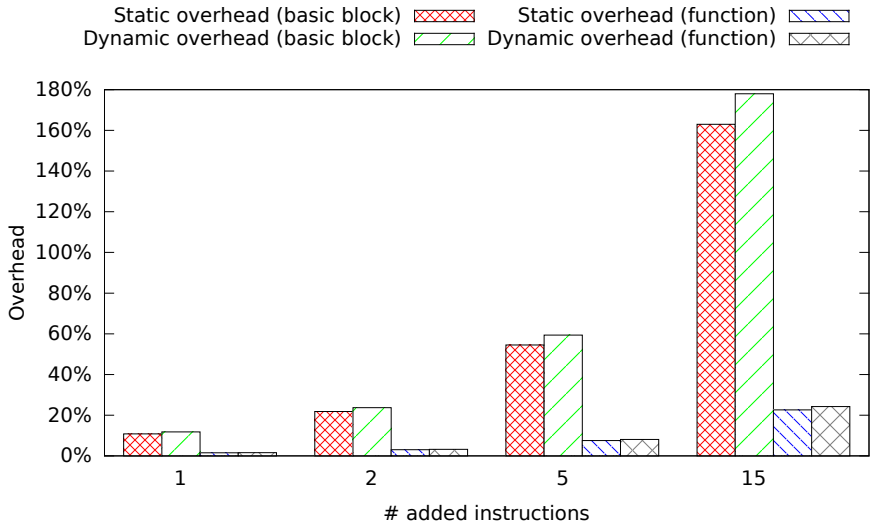


Figure 7.6: Injection static and dynamic overhead of injecting instructions both at the basic block level and the function call level

is: if a detector is retrained with the addition of sample evasive malware, would it be able to classify them correctly? If the answer is yes, then perhaps the weights can be updated regularly to allow the detector to adapt to emerging malware. However, there is still a possibility that the retrained detector could itself be reverse-engineered and evaded again. Moreover, as attackers continue to evolve, it is not clear if the detector will eventually become ineffective due to the number of classes it is attempting to separate, or if it will converge to a classification setting that is impossible to evade.

Figure 7.8(a), shows the effect of increasing the percentage of evasive malware programs in the training data that we use to retrain the simple LR detector. For example, the point with 10% indicates that 10% of the malware part of the training set consists of evasive malware modified with one of our evasion strategies. We see that in general, increasing the percentage of evasive malware leads to more accurate detection. Unfortunately, this comes at the price of loss of accuracy (sensitivity) for non-evasive malware, making simple retraining an ineffective strategy. It's interesting to note that the correct classification of normal programs (specificity) does not suffer.

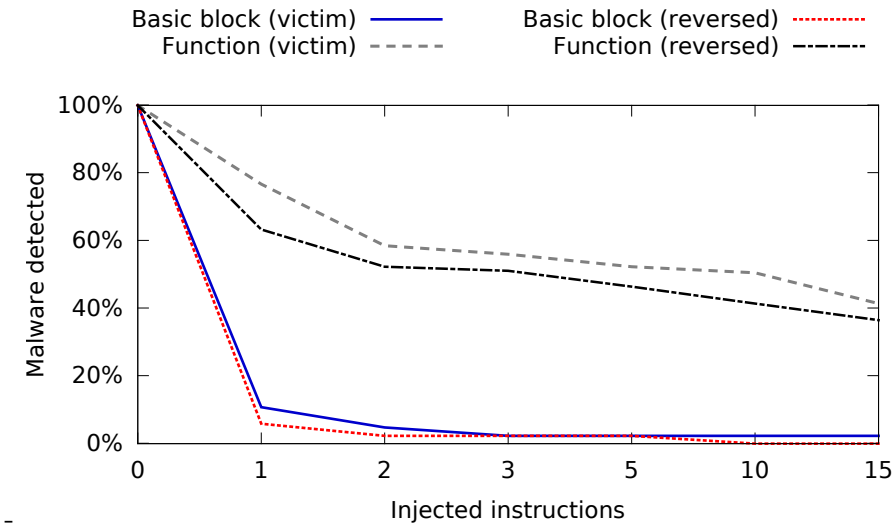
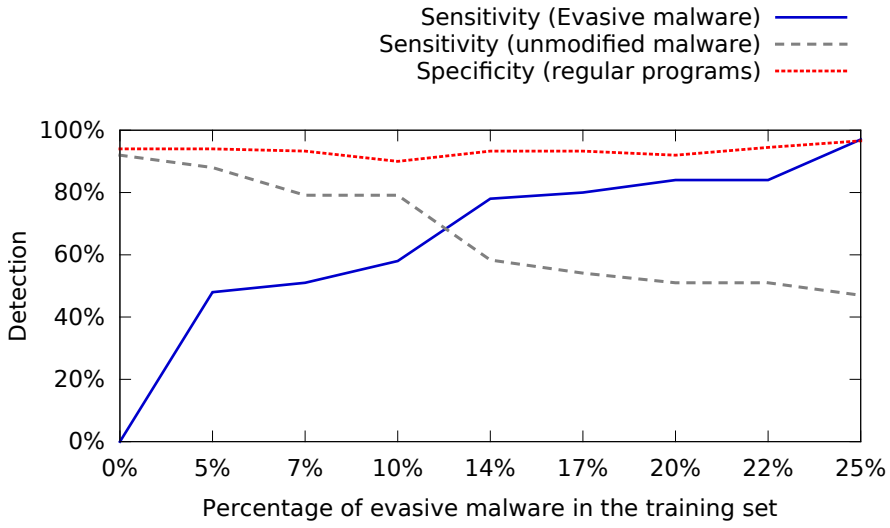
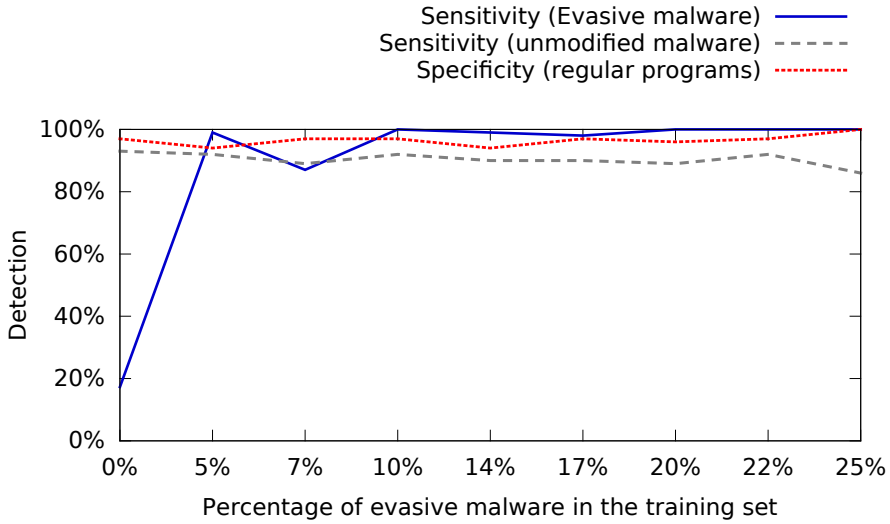


Figure 7.7: Detection performance of evasive malware using the weighted injection method.

Figure 7.9(a) illustrates why linear detectors such as LR have to sacrifice accuracy when retrained. Figure on the left shows that there is a linear separation between the malware and regular programs. Figure in the middle demonstrates that in order to evade detection, the evasive malware have to cross the separation boundary. Figure on the right shows that with retrained detector it may be impossible to find a linear separation between malware (including evasive malware) and regular programs. In contrast, non-linear classifiers such as NN (Figure 7.8(b)) are able to detect this new form of malware with high accuracy even with a low percentage of evasive malware in the retraining set. This can be achieved without affecting the detection accuracy of non-evasive malware or regular programs. Figure 7.9(b) illustrates why non-linear detectors are more effective when retrained. Even when evasive malware crosses the separation boundary of the original classification, a new non-linear boundary can be found that separates the two malware classes from normal programs. Thus, HMDs must be non-linear if we want to allow them to be retrained in response to evasive malware.



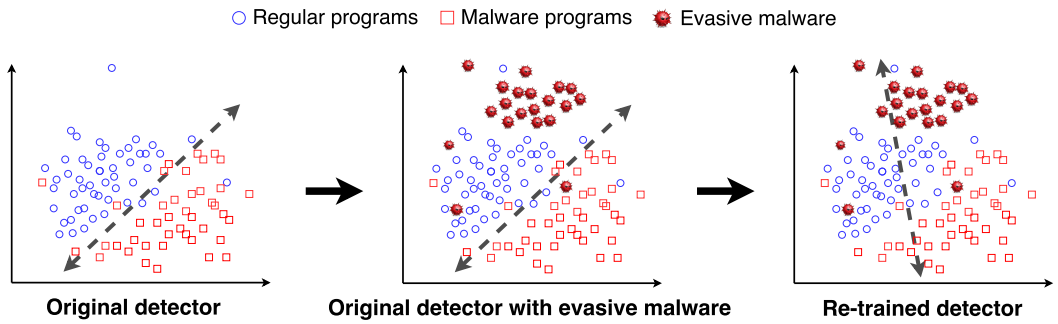
((a)) Logistic regression



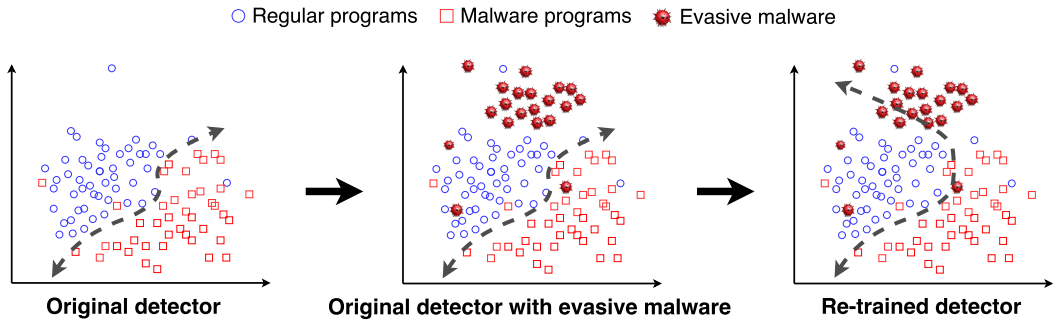
((b)) Neural networks

Figure 7.8: Effectiveness of retraining; can retraining the detectors with evasive samples detect evasive malware?

Figure 7.10 shows the detection of several generations of NN detectors. In each generation, we repeat retraining of the detector by adding malware from the previous generations to the training set. The original detector in generation 1 is first evaded successfully as we see low detection for evasive malware. After we retrain, we see that evasive malware developed to evade detector 1 is now detected successfully (rightmost bar for detector 2). However, if we reverse-engineer the detector and evade it again, we



((a)) Linear model



((b)) Non-Linear model

Figure 7.9: Illustration of effect of retraining a linear and non-linear classifiers with data that includes evasive malware.

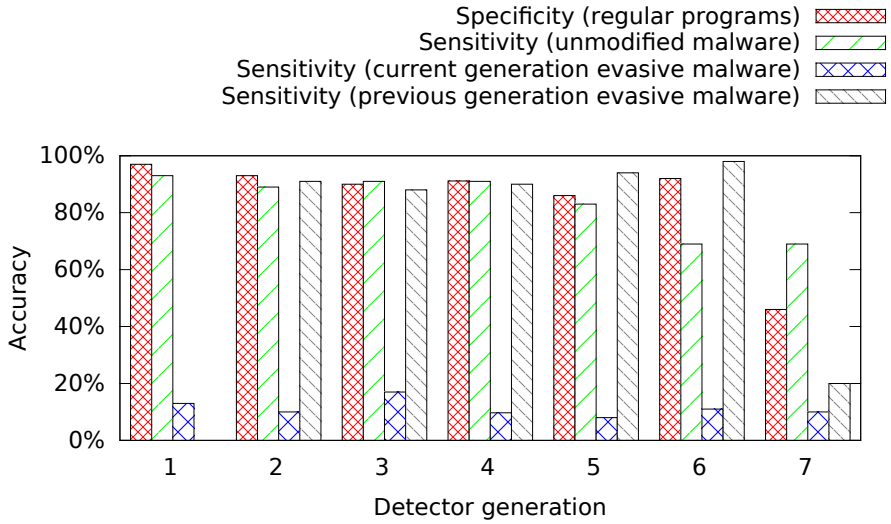


Figure 7.10: Evasive malware detection performance of NN detector for each retraining on evasive malware generation.

can do so successfully as evidenced by the low detection of the evasive malware in the third bar for detector 2. As the retrain-evade process is continued, we expected one of two outcomes: (1) the detector will no longer be able to classify; or (2) the decision

boundary will tighten and malware will no longer be able to evade. The second outcome occurred: after 7 generations, the detector can no longer be trained successfully as malware and normal programs became inseparable using our NN. There are two possible explanations: (1) the feature is not sufficiently discriminative, and its possible to turn malware to be similar to normal programs with respect to this feature. Note that in each successive generation, the overhead is increased, and this level of overhead may not be acceptable to the attacker; or (2) NN could no longer represent the complex decision boundary between the different classes of evasive malware and normal programs, similar to how LR failed after one generation.

Chapter 8

Evasion-Resilliance Hardware Malware Detectors (RHMD)

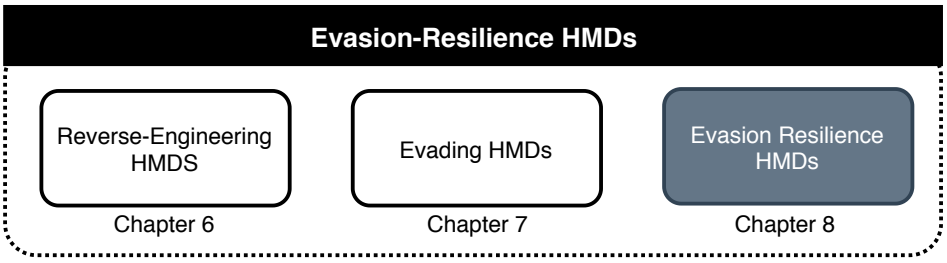


Figure 8.1: Building evasion-resilience HMDs project overview

After showing that the current generation of HMDs is vulnerable to evasion, in this chapter, we explore whether new HMDs can be constructed that are robust to evasion. In particular, we propose in Section 8.1 a new resilient HMD (RHMD) organization that uses multiple diverse detectors and switches between them unpredictably. We show that RHMDs that use even simple base detectors are resilient to both reverse-engineering and evasion. Furthermore, this resilience increases with the number and diversity of the base detectors.

In addition, in this chapter, we explore whether RHMDs fundamentally increase the difficulty of evasion or simply present another hurdle that can be bypassed by attackers. To this end, in Section 8.2 we overview recent results in Probably Approximately Correct (PAC) learnability theory that proves that RHMDs provide a measurable advantage in increasing the difficulty of reverse-engineering and complicate evasion. By making HMDs resilient to evasion, we bring them closer to practical deployment.

In summary, in this chapter, we develop a new class of evasion-resilient HMDs (RHMDs) that operates by randomizing detection responsibility across different diverse detectors. RHMDs cannot effectively be reverse-engineered to enable evasion, which we support both experimentally and using recent results from PAC learnability theory. The number and diversity of the individual classifiers used increases the resilience to reverse-engineering and evasion. In addition, we study implementation complexity of such classifiers in hardware.

8.1 Evasion-Resilient HMDs

Although retraining the detectors can allow the updated detectors (retrained detectors using evasive malware samples) to detect evasive malware that has representatives in the training set, we showed retraining may eventually fail as attackers continue to evade. Moreover, retraining cannot detect novel evasive malware: even after retraining they can be reverse-engineered and evaded.

In this section, we introduce a new class of evasion-resilient HMDs (RHMDs). RHMDs leverage randomization to make detectors resistant to reverse-engineering and therefore evasion. In particular, this is a strong advantage in the sense that randomization introduces an error to the reverse engineering that is bounded by a function of

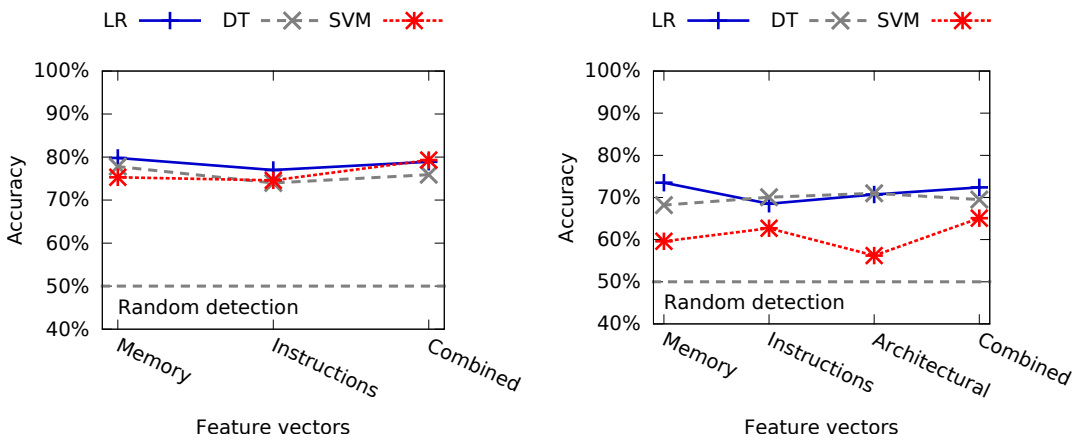
how often the detectors disagree; we show a proof for this claim in Section 8.2 based on PAC learnability theory.

We randomize two settings of the detectors: i) The feature vectors used for detection; and ii) The collection periods used in the detection. In particular, we construct detectors with these heterogeneous features and switch between them stochastically in a way that cannot be predicted by the attacker.

Our first study examines the effect of randomizing the feature vectors used for detection. We start with two detectors using the same detection period. The results of this experiment are shown in Figure 8.2(a). We reverse-engineer the detector using two of the original feature vectors as well as a combination of them. In particular, the point in the figure marked "combined" represents reverse-engineering with a combined detector using the union of the two feature vectors. Using an RHMD with two detectors, reverse-engineering the detector becomes substantially more difficult because the model now includes two diverse detectors which are selected randomly. The diversity can be further expanded by using a pool of three detectors — the results of this approach are shown in Figure 8.2(b). Again, the combined point on the figure refers to a reverse-engineering attempt using the union of the three feature vectors of the three individual detectors. As seen from the results, reverse-engineering becomes harder with increased diversity.

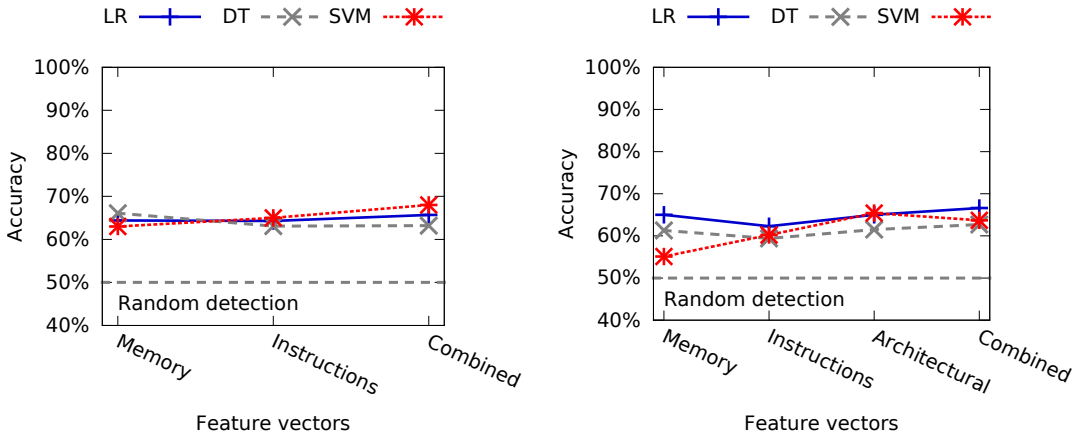
To further increase detector diversity, we construct detectors with two different collection periods (10K cycles period and 5K cycles period), resulting in a pool of six detectors, which are randomly chosen by the detection logic. The results are presented in Figure 8.3. Consistent with the previous trend, additional diversity makes reverse-engineering even more difficult. Note that having detectors operating on the same features with different period does not substantially increase the hardware complexity;

the different weight for the two detectors must be kept separately, but the collection logic and the detector evaluation logic is shared.



((a)) Two feature vectors ((b)) Three feature vectors

Figure 8.2: Reverse engineering RHMD that have different features vectors for base detectors.



((a)) Two feature vectors ((b)) Three feature vectors

Figure 8.3: Reverse engineering RHMD that have different features vectors and detection periods for base detectors.

Having reverse-engineered the detector, we use our evasion framework to inject instructions to evade it. Given that the reverse-engineering becomes inaccurate in RHMDs and given the random switch between the individual detectors, the constructed evasive malware can no longer hide from detection (Figure 8.4). It is interesting to note

that the higher the diversity of the detector, the more resilient it is to evasion, consistent with PAC learnability theory discussed in the next section. These results demonstrate that this approach to constructing HMDs provides resilience to evasive malware. The average detection accuracy of the RHMD without evasion (Figure 8.4 with 0 injected instructions) is equal to the average accuracy of its base detectors since the randomization selects between the detectors with equal probability. Thus, the average loss of detection due to randomization is the difference of accuracy between the most accurate detector and the average of all base detectors.

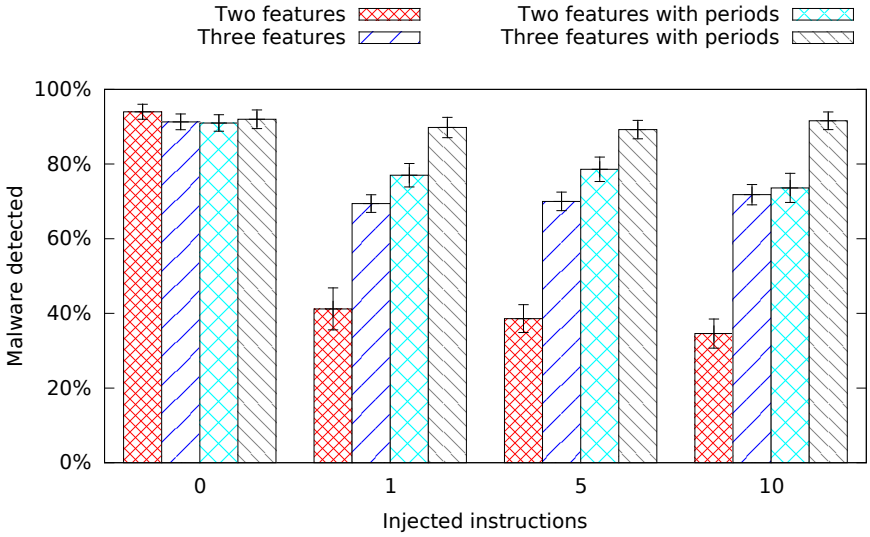


Figure 8.4: RHMD evasion resilience; performance of detection evasive malware using different configurations of RHMDs.

To evaluate the overhead of implementing the detectors for online detection [122], the proposed resilient detectors were implemented using Verilog, as an extension of an open source x86-compatible core (AO486) [4] to estimate the overhead. The detectors collect information from the commit stage of the pipeline and apply the detection logic at the detection period. After synthesizing the new core implementation on an FPGA board for a configuration with three detectors corresponding to the three features with the same period, we observed that the area and power increase is modest: 1.72% and

0.78% respectively. Note that the resilient detectors can also be used to make offline detection [38] resilient to evasion.

8.2 Theoretical Basis for RHMD

This section provides theoretical support for the resilience of RHMD for evasion based on probably approximately correct (PAC) learnability theory [**?**]. In particular, we show that randomized classification is inherently more difficult to be reverse-engineered than a deterministic classifier (even one with arbitrarily high complexity).

8.2.1 Learnability of Deterministic Classification

Consider a learning system (a defender) that uses a single classifier to detect malware from normal programs. Consider also another *reverse engineering* learning system (an attacker) that uses data of past classification collected, for example, by repeatedly querying the defender classifier, to determine with high accuracy the nature of the defender classifier.

Formally, let H be the class of possible classifiers (also called hypothesis class) a learning system considers. Let P be the probability distribution over instances (x, y), where x is an input feature vector, and y is a label in $\{0, 1\}$. We assume below that $y = \bar{h}(x)$, i.e., a deterministic function that gives the true label of x. For any $h \in H$, let $e(h) = \Pr_{x \in P}[h(x) \neq \bar{h}(x)]$ be the expected error of h w.r.t. P. We define $e_H = \inf_{h \in H} e(h)$ as the optimal (smallest) error achievable by any function $h \in H$. Let $D = \{(x_1, y_1), ..., (x_m, y_m)\}$ be a training data set of size m generated according to P and $\mathcal{D}$ be the set of all possible D. A learning algorithm is a function $L : \mathcal{D} \rightarrow H$ which produces a classifier $\hat{h} \in H$ given a training set D.

Definition 1 *A hypothesis class H is learnable if there is a learning algorithm L for H with the property that for any $\epsilon, \delta \in (0, 1/2)$ and distribution P there exists a training sample size $m_0(\epsilon, \delta)$, such that for all $m \geq m_0(\epsilon, \delta)$, $\Pr_{D \in \mathcal{D}}[e(L(D)) \leq e_H + \epsilon] \geq 1 - \delta$, i.e., L will with probability at least $(1 - \delta)$ output a hypothesis $\hat{h} \in H$, whose error on P is almost $(e_H + \epsilon)$. H is efficiently learnable if $m_0(\epsilon, \delta)$ is polynomial in $1/\epsilon$ and $1/\delta$, and L runs in time polynomial in m, $1/\epsilon$ and $1/\delta$.* [1]

The definition of PAC learnability above says that a hypothesis class H is efficiently learnable (i.e., learning is easy) if we can compute with high probability an approximately optimal candidate from this class given a polynomial number of samples. The error bound $(e_H + \epsilon)$ for an approximately correct classifier $\hat{h} \in H$ consists of two components. ϵ becomes arbitrarily small and hence $e(\hat{h})$ approaches e_H when the number of training samples increases polynomially w.r.t. $1/\epsilon$. The other component e_H depends on the learning bias [**?**] about H. That is the set of assumptions that the learner makes (e.g., the type of classifiers and the underlying features). With a good choice of H, e_H can be arbitrary small; e_H is zero if H contains the true classifier $\bar{h}$. We observed the implications of this result in Section 6.3 when the correct feature and detection period lead to the highest accuracy reverse engineered classifier.

The concept of PAC learnability applies to the learning tasks by both the defender and the attacker, with one caveat: for the defender, the goal is to correctly predict the *true* label of an instance (i.e., $y = \bar{h}(x)$); while for the attacker, the goal is to correctly predict the label of an instance *assigned* by the defender's classifier (i.e., $y = \hat{h}(x)$). As shown in [162], efficient learning for $\bar{h}$ by the defender implies efficient learning for $\hat{h}$ (called efficient reverse-engineering) by the attacker. Suppose that the defender has learned $\hat{h}$ from an efficiently learnable H. Provided the attacker identifies

[1] This definition is taken, in a slightly extended form, from [**?**].

the type and features of the classifiers in H, then $\hat{h}$ is contained in the hypothesis class used by the attacker, and $e_H = 0$, i.e., the distribution P over $(x, \bar{h}(x)$ can be efficiently reverse-engineered with arbitrary precision [162]. Without prior knowledge of H, the attacker can tune its hypothesis class based on the error rate on the training data collected over repeated queries.

These results support the reverse-engineering experiments in Section 6.3. In particular, the analysis shows that *reverse-engineering a deterministic classifier is "easy" in practice regardless of the complexity of the defender's classifier*. Increasing the complexity of the defender's classifier can make it more costly to reverse-engineer it, but will not change the outcome of the arms race between the defender and attacker with respect to the difficulty of evasion.

8.2.2 Learnability of Randomized Classification

We now consider a defender that uses randomized classification such as the model used in RHMD. As before, consider a distribution P over instances (x, y), with $y = \bar{h}(x)$ as the ground truth. Suppose that we have n hypothesis classes H_i, all efficiently learnable. Let $\hat{h}_i \in H_i$ be the classifier learned from these classes, respectively, with the corresponding error rate $e(\hat{h}_i)$. Additionally, let $\triangle_{i,j} = \Pr_{x \in P}[\hat{h}_i(x) \neq \hat{h}_j(x)]$ for all i, j: that is, $\triangle_{i,j}$ measures the difference between two classifiers $\hat{h}_i(x)$ and $\hat{h}_j(x)$ over the data distribution. Consider a space of policies parametrized by $p_i \in [0, 1]$ with $\sum_i p_i = 1$, where we choose $\hat{h}_i \in H_i$ with probability p_i. Let $\vec{p}$ denote the corresponding probability vector. Then, a policy $\vec{p}$ induces a distribution $Q_{\vec{p}}$ over (x, z), where $z = \hat{h}_i(x)$ with probability p_i. The defender will incur a baseline error rate of $e_{\vec{p}}(h) = \Pr_{x \in Q_{\vec{p}}}[\hat{h}_i(x) \neq \bar{h}(x)] = \sum_i p_i e(\hat{h}_i)$ if there is no reverse-engineering effort.

Suppose the attacker observes a sequence of data points from $Q_{\vec{p}}$, and tries to efficiently learn the hypothesis class $H = \cup_i H_i$. For any $h \in H$, let $e_{\vec{p}}(h) = \Pr_{x \in Q_{\vec{p}}}[h(x) \neq \hat{h}_i(x)] = \sum_i p_i e(h)$, the expected error of h w.r.t. $Q_{\vec{p}}$, and we define $e_{\vec{p},H} = \inf_{h \in H} e_{\vec{p}}(h)$ as the optimal (smallest) error achievable by any function $h \in H$ under a policy $\vec{p}$. Definition 1 naturally extends to the randomized setting: in particular, the distribution P becomes $Q_{\vec{p}}$ and the error bound $(e_H + \epsilon)$ becomes $(e_{\vec{p},H} + \epsilon)$.

Theorem 1 *Suppose that each H_i is efficiently learnable, and $\hat{h}_i \in H_i$ be the classifier learned from these classes by a defender, respectively, with the corresponding error rate $e(\hat{h}_i)$. Then, any distribution $Q_{\vec{p}}$ over (x, z) can be efficiently reverse engineered, with $e_{\vec{p},H}$ bounded by $\min_i \sum_{j \neq i} p_i \triangle_{i,j} \leq e_{\vec{p},H} \leq 2(\max_i e(\hat{h}_i))$.* [2]

This theorem shows that on the one hand, even with randomization, reverse-engineering is easy as long as all classifiers among which the defender randomizes accurately predict the target - that is $\max_i e(\hat{h}_i)$, the maximal error among the n classifiers, is arbitrarily small. On the other hand, the attacker's error depends directly on the difference among the classifiers, which can be significant if at least some of the classifiers are not very accurate, allowing them to disagree more often. According to the error bound $(e_{\vec{p},H} + \epsilon)$, even though ϵ becomes arbitrarily small as the number of queried samples increases, the defender will inevitably suffer from an error caused by $e_{\vec{p},H}$. This error can be high; for example, when randomizing two classifiers of error 0.2 and 0.1 with $p_1 = p_2 = 0.5$, $e_{\vec{p},H}$ is in [0.15, 0.4]. In contrast, in the deterministic setting, e_H can be 0. For our experiments with the pool of six detectors we measured the error to be around 25% on our testing dataset.

[2] This theorem is formed by combining Theorem 2.2 and Corollary 2.3 (with detailed proofs) from [162].

The above theorem also suggests a trade-off between the accuracy of the defender under no reverse-engineering vs. the susceptibility to being reverse-engineered: using low-accuracy but high-diversity classifiers allow the defender to induce a higher error rate on the attacker, but will also degrade the baseline performance against the target. To combat reverse engineering effort, we propose to randomize among a set of low-complexity, low-accuracy classifiers (e.g., logistic regression), rather than deploying a single high-complexity, high-accuracy classifier (e.g., deep neural network or random forest). The former is also more suitable for a hardware implementation than the latter. Although this low accuracy applies to the classification of each individual period, we raise the overall accuracy of the detector by averaging the decisions across multiple intervals.

8.2.3 Evasion Without Reverse Engineering

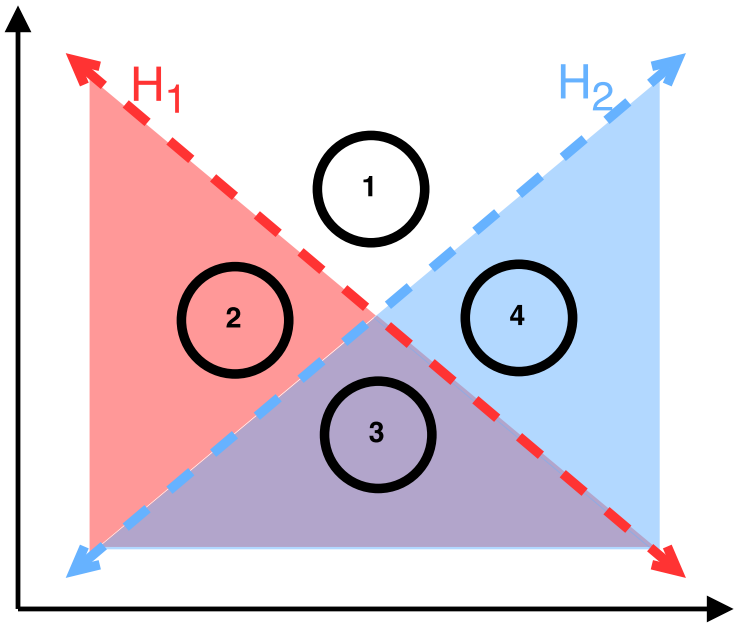


Figure 8.5: Impact of Randomization on Evasion

Our threat model assumes that an attacker needs to reverse engineer a detector before evading it. The theoretical resilience claims on RHMDs rely on the difficulty of this reverse engineering. In this section, we consider whether it is possible to evade the detector without reverse engineering. To provide intuition, we start with Figure 8.5, which shows the decision boundaries of two diverse base detectors learned from hypothesis classes H_1 and H_2. The two decision boundaries are not mutually exclusive (H_1 malware regions are 2, 3 and H_2 malware regions are 3, 4). To fool both detectors, the malware has to move to region 1 which both detectors treat as normal. Note that these decision boundaries represent hyperplanes in an n-dimensional feature space for LR, and complex surfaces in the same space for NN. Therefore, as we increase diversity the target area for evasion gets smaller. Thus, provided that detectors are diverse, making random insertion guesses is unlikely to succeed and expensive to validate. Note that evasion must succeed continuously across consecutive detection windows, which complicates attempts to incrementally evade the detector.

This example also provides intuition on why randomization complicates reverse engineering (as shown by Theorem 1). The attacker has to suffer a significantly increased error ($e_{\vec{p},H}$) if she tries to learn a decision boundary from the same hypothesis classes adopted by the defender. Otherwise, she has to learn a decision boundary of a higher complexity class which requires exponentially larger number of samples.

If the attacker knows precisely the configuration of the base detectors of an RHMD, we verified that it is possible to evade it, for example, by iteratively evading each. This approach incurs a high overhead since instructions need to be injected to evade each of the detectors. We do not consider this case as part of our threat model. Resilience in this case may be achieved if we make the decision boundary of the RHMD non-stationary. This can be accomplished by having a large set of candidate features

and periods, of which a random subset is used for the RHMD at any given time. This is an interesting area for future research.

Chapter 9

SafeSpec: Leakage-Free Speculation

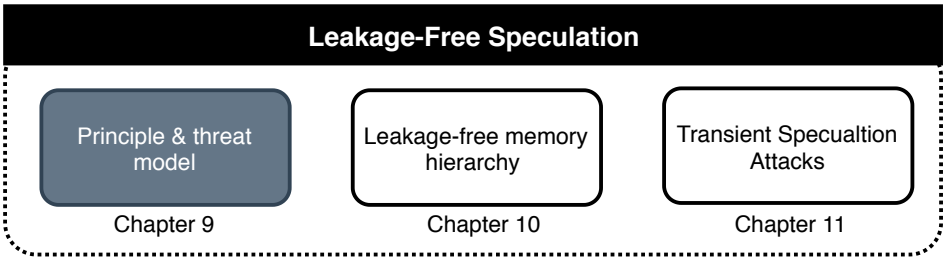


Figure 9.1: Leakage-free speculation project overview

Speculative execution is a standard microarchitectural technique used in virtually all modern CPUs to improve performance. Recently, it has been shown that speculatively executed instructions can leave measurable side-effects in the processor caches and other shared structures even when the speculated instructions do not commit and their direct effect is not visible. The recent Meltdown and Spectre attacks [98, 89, 58, 106, 93, 105, 160, 167, 29, 151, 9, 14, 140] (we call this class of attacks *speculation attacks*) have shown that speculation can be exploited to expose information that is otherwise inaccessible. In a typical scenario, attackers either mis-train the branch predictor

unit or directly pollute it [46, 48] to force the speculative execution of code that reads privileged data (privilege checks are not enforced during speculation). Although the speculative instructions will eventually get squashed, leaving no direct data accessible to the attacker, they leave a side-channel trail that can be used to infer the value. Several attack variations have been demonstrated, including arbitrary exposure of the full memory of other processes, OS kernel, hypervisor, and even SGX enclaves [29, 160] to an unprivileged attacker, making this a dangerous open attack vector on modern systems. We describe these attacks and present our threat model in Section 9.1.

Although a number of defenses and software patches have been proposed to mitigate Spectre and Meltdown [159, 53], they often address only one aspect of the attack, leaving attackers with other possible variations that are still available. In addition, these patches often lead to high overheads: 10-30% reported on average, but often much higher. For example, Netflix reported 800% slowdown with the Meltdown patches on their systems [158, 52]. Most of the solutions target a subset of the threat models and make assumptions that can be broken by future architectures.

In this project, we explore whether speculation can be made leakage free in a principled way, enabling CPUs to retain the performance advantages of speculation while removing the security vulnerabilities that speculation exposes. To this end, we introduce *SafeSpec*, a design principle where speculative state is stored in temporary structures that are not accessible by committed instructions. As instructions transition from being speculative to committable, any speculative state is moved to the permanent structures. On the other hand, if a speculative instruction is squashed, the speculative side effects are canceled in place leaving no measurable side effects in the permanent structures and closing the vulnerability exploited by speculation attacks. We consider two variants that differ in when an instruction is considered safe to commit; In the wait-for-commit

(WFC) variation, we consider an instruction speculative until it is committed. In the wait-for-branch (WFB) variation, we consider an instruction to be committable when the last control flow instruction it depends on commits. Note that only WFC prevents Meltdown-style attacks which do not depend on a branch misspeculation, but it is possible that other defenses can cover Meltdown since only Intel processors appear to be vulnerable to it.

SafeSpec makes no assumptions on the branch predictor behavior or on speculative execution behavior; for example, it does not prevent the attackers from mis-training or even polluting the branch predictor, nor does it prevent them from speculatively reading privileged data. Rather, SafeSpec interferes with the attacker's ability to create a covert channel using speculative data accesses to communicate illegally-accessed data out. We describe *SafeSpec* in Section 9.2.

In summary, in this chapter, we introduce the *SafeSpec* model to protect speculation by isolating speculative state from committed state.

9.1 Speculation Attacks and Threat Model

In this section, we introduce speculation based side-channel attacks (speculation attacks for short). The section first discusses speculative execution to characterize the capabilities of the attacker, and then overviews the Meltdown and Spectre attacks.

9.1.1 Speculative Execution in Modern Processors

Speculative execution has been an important part of computer architecture since the 1950s. The IBM Stretch processor implemented a predict not-taken branch predictor [19]. As computer architecture continued to advance rapidly, the amount of

speculation that is exploited has progressively been increasing with aggressive out-of-order execution, supported by sophisticated branch predictor designs [176, 72, 144] that are highly successful in predicting both the branch direction and its target address. In particular, the number of pipeline stages in production CPUs has continued to grow to the point where modern pipelines commonly have between 15 and 25 stages. Moreover, with out-of-order execution, when a branch instruction stalls (e.g., due to a cache miss on which it depends), instructions that follow the branch are continuously being issued. Thus, the speculation window can be extremely large, typically limited by the size of structures such as the reorder buffer, which can hold a few hundred instructions.

Speculation is designed to not affect the correctness of a program. Although branch mispredictions occur and speculative instructions can ignore execution faults (e.g., permission error for memory access) these semantics were not considered harmful as mis-speculation will eventually be detected and the uncommitted instructions will be squashed, leaving no directly visible modifications to the architectural state such as registers and memory. Micro-architectural structures such as caches and TLBs are affected by speculative operations, but the contents of the cache only affect performance, not the correctness of a program. In fact, prior work has shown that there are beneficial prefetching side-effects to speculatively executed instructions even those that are eventually squashed [112]. To exploit these effects, designs such as runahead execution [113] intentionally increase the speculation window beyond the physical limitations of the reorder buffer to generate additional cache misses further into the program to exploit their prefetching effect. This approach was shown to significantly improve single-threaded performance.

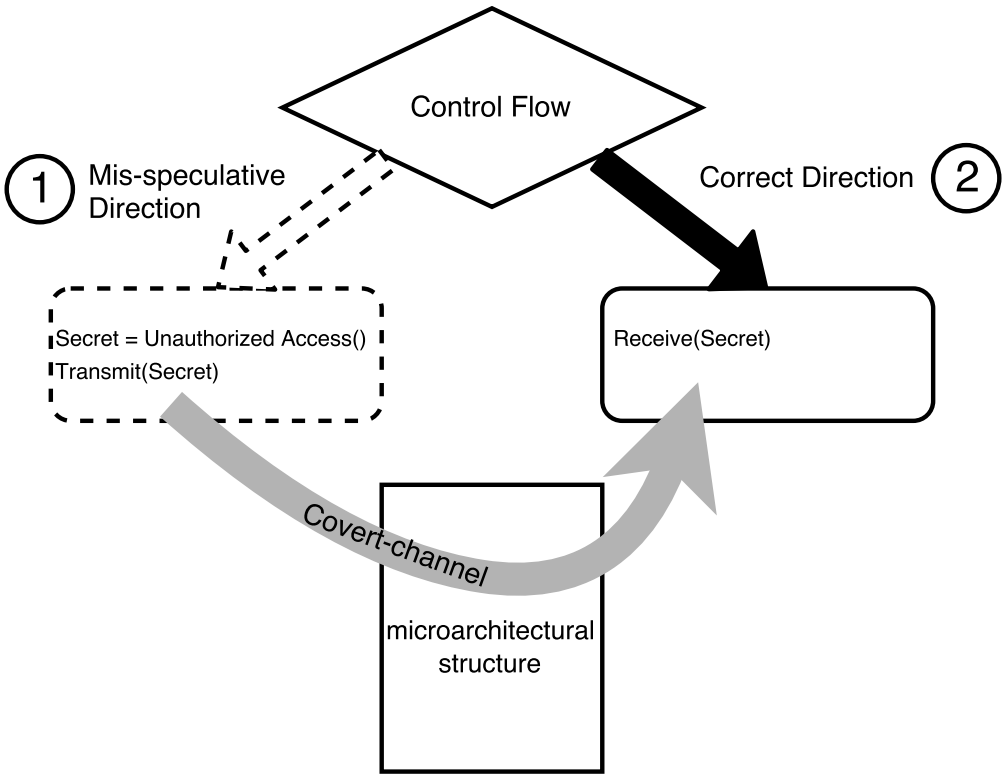


Figure 9.2: Overview of speculation attacks.

9.1.2 Speculation Attacks

Meltdown and Spectre are two representative attacks of the class of speculation attack. In general, these attacks exploit three properties of speculative execution in modern processors:

- **P1**: branch prediction validation and permission checks are performed deep in the pipeline and execution fault is generated only if the instruction is committed, enabling speculative instructions to access data outside its privilege domain;
- **P2**: speculative instructions leave side-effects in micro-architectural structures such as caches, which can be inferred using well-known techniques like Flush+Reload [174] and Prime+Probe side-channel attacks [100].

- **P3**: the branch predictor can be mistrained (Spectre 1), or directly polluted (Spectre 2). It is shared across all programs running on the same physical core [58, 89, 46], allowing code running in one privilege domain to manipulate branch prediction in another domain (e.g., kernel, VM, hypervisor, another process, or SGX enclave).

Next, we overview how different variants of Spectre and Meltdown attacks work, and distinguish them based on how they trigger and leverage speculative execution.

```
unsigned char secret;
dummy  = array[secret * 64];
```

Figure 9.3: Secret-revealing gadget.

A Common Gadget

Speculation attacks aim to "read" memory/register content that is otherwise restricted. Unlike traditional memory reads, speculative reads are based on triggering speculative execution of a small code chunk, called gadget. A simplified example of such gadget is demonstrated in Figure 9.3. Assume the variable secret holds a secret value and is used as an index into a byte array. If processor speculatively executes this code, a memory access will be generated, and as a result some data will be borough into the data cache. Note that the secret variable controls what cache set will be updated by the speculative execution hardware. The multiplication operation ensures that different values of the variable will result in different cache sets. Because in many cases the location of data and code structures in victim process memory is not secret, the attacker, capable of monitoring CPU cache activity can link the observed behaviour with the corresponding value of the secret variable. For example, by knowing cache

set s_0 is accessed when the value of secret is 0, the attacker can deduce that the value of secret must be $1(\mathrm{mod}n)$, with n equals to the number of sets in the cache, when an access to the cache set s_1 is detected. Cache updates can be detected by attacker using a range of cache side channel attacks [174, 100, 54]. Please note that in normal execution, this code will never be executed, otherwise it will result in trivial cache side channel leakage. In speculative attacks the attacker uses the properties **P1** and **P3** described above to trigger the gadget to be *speculatively* executed by the victim.

```
if (offset < array1_size)
    y = array2[array1[offset] * 64];
```

Spectre (Variant 1)

This variant of the attack can be demonstrated by the code presented in Figure **??**. In this code, a victim process reads values from array1 using the offset provided by the attacker. Then, resulting value is used to perform an access into array2. As we discussed above, accesses into the array2 can be used by the attacker to deduce the value of the index. The index, in its turn, is controlled by the attacker since attacker controls the offset. Therefore, the attacker can use a carefully selected value of offset to read arbitrary memory address which then will result in cache access observable by the attacker. However, the if statement ensures there are no out of bounds memory accesses allowed. Unfortunately, the attacker can exploit speculative execution and behavior of branch predictor to force the victim process to perform an out of bounds memory access in the following way:

a) The attacker triggers the code to be executed several times with the value of the offset such that the if statement is always true *(branch instruction not-taken)*. This trains the branch predictor to predict the corresponding branch always not-taken;

b) Next, the attacker flushes array1_size from the cache, forcing the CPU to fetch the value of array1_size from memory, delaying the correct evaluation of the branch and creating a large speculative window;

c) Finally, the attacker provides the malicious offset. The branch predictor unit predicts the branch not-taken, resulting in two memory accesses that reveal the value stored at the attacker's desired address.

Spectre (Variant 2)

In this variant of the attack, the target program may not contain the expected gadget or the offset is not controllable by the attacker. These limitations can be bypassed by *hijacking* the speculative execution. Specifically, when the CPU encounters an indirect branch instruction, the branch predictor tries to guess the target address and the CPU immediately starts speculatively executing instructions at this address. Due to **P3**, the attacker can perform the *branch target poisoning* to hijack the speculative execution flow and to redirect it to any code location containing gadget instructions. This resembles the return-oriented programming attack [28]. In summary, the variant 2 attack works as follows:

a) The attacker ensures that the attacking code and the victim code share the same branch target buffer (BTB) by executing attacker's process on the same physical core with victim.

b) The attacker forces a BTB collision by matching virtual address of the victim and attacker branch instructions [46].

c) The attacker performs target poisoning by executing its own branch.

d) Finally, the attacker triggers the indirect branch to be speculatively executed, redirecting the speculative execution to a gadgets of attacker's choice. The gadget will leak data through a side channel in a way similar to previously described.

Meltdown

Meltdown attack exploits **P1**: due to pipelining and instruction reordering a permission check can happen *after* the corresponding memory accesses is speculatively executed. For example, assume a user application that tries to read kernel memory. Although such request will be eventually denied, the speculatively executed instructions will result in loading of requested data into caches. Using a side channel, the attacker can effectively read arbitrary kernel (or hypervisor) memory. This is a very powerful attack, since typically kernel memory contains a direct mapped region allowing the attacker to dump the entire physical memory on a given system. Since the exception eventually will be raised, this attack requires the ability to tolerate and recover from segmentation faults. Alternatively, if the attacker can arbitrarily control the exploit code, she can also avoid the exception by putting the gadget behind a mispredicted branch, i.e., combining Spectre V1 with Meltdown to read memory across privilege domains in the same virtual address space.

9.1.3 Threat Model

Since **P2** is essential in all speculation attacks, this work aims to eliminate the side-effects from speculative execution. Hence, we assume a strong adversary for the branch predictor and no software-based defense for branches (e.g., lfence and retpoline [159]). In particular, we assume that attackers can arbitrarily control the state of the branch predictor, as if its state is programmable without any privilege. We as-

sume attackers can launch a speculation attack either from the same process or another process. We assume attackers have complete control over the attacking code (as in the Meltdown attack) and know the complete layout of the victim domain (another process, kernel, enclave, etc.). Their goal is to reveal memory and/or register content of the victim domain. We assume the victim domain does not have any direct channel or vulnerability to leak the content so attackers must utilize side-channels. To enable these side-channel, we assume the victim domain contains the code gadget such as the one in Figure 9.3 that can be invoked by attackers.

The technical solution we propose is general and applicable to different micro-architectural structures. However, as a demonstration, our prototype implementation only protects caches and TLBs to explore concretely the implications and complications that result from *SafeSpec*. Therefore, we further assume that other covert channels, including the ones through the branch predictor, memory bus and DRAM buffers are out-of-scope for the current paper, but will be addressed using similar principles by future work. Similarly, we only consider a system with a single core. Thus, speculation attacks against the cache coherence and memory consistency model states [157] are also left for future work. We discuss the implications on both of these in Section **??**.

9.2 *SafeSpec*: Leakage-free Speculation

SafeSpec is a principled approach to secure processors against speculation attacks while retaining the ability to carry out speculative execution to benefit from its performance. The general principle (shown in Figure 9.4) addresses the problem at the root by introducing shadow state to separate state that is produced speculatively without affecting the primary structures of the processor (which we call committed state).

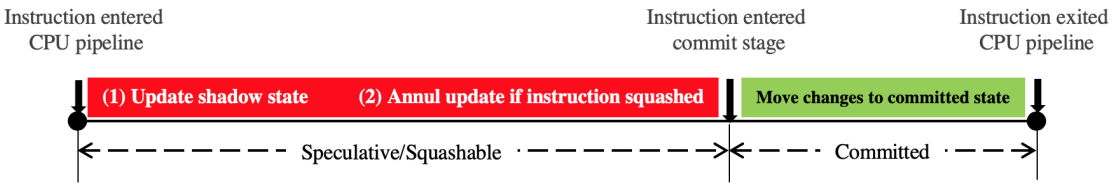


Figure 9.4: SafeSpec overview

For example, if a speculative load instruction causes a load of a cache line, instead of loading that cache line into the processor caches, we hold the line in a temporary structure. If the load instruction is later squashed, these effects are removed in place, leaving no changes to the cache from the misspeculated instructions, and closing the vulnerability. Alternatively, if the instruction commits, the cache line is moved from the temporary structure to the L1 cache. While is simple in principle, a number of questions relating to its security, complexity and performance have to be resolved.

When to move state from speculative to committed. There are two options available to decide when to move state from the shadow to the committed state. In the first variation, which we call *wait-for-branch* (WFB), we can assume an instruction to be no longer speculative when all the branches (more generally, all predictions) it is dependent on have been resolved. WFB stops all variants of spectre which depend on mistraining the branch predictor/return stack buffer; none of the mis-speculated instructions moves to the committed state. However, it does not prevent Meltdown which relies on speculation within a single instruction.The second variation *wait-for-commit* (WFC) waits until the instruction commits before moving its effects to the committed state, and therefore also prevents Meltdown.

Shadow state organization and size: If the shadow state structures are too small, then either speculative state is replaced (causing a loss of an update to the committed state if this data were to be committed later), or the instruction has to stall until there

is room in the speculative structure before it issues. From a performance perspective, the organization and size of the shadow structure should be designed such that the structures can hold the speculative state generated by speculation as measured across typical workloads. However, we will show that security considerations introduce more stringent requirements on the speculative state.

Filtering Delayed Side Effects: One of the issues with *SafeSpec* occurs when an instruction is squashed in the middle of its execution. If the instruction has already initiated a high latency operation such as a read from memory, we have to ensure that the response from memory can be discarded after it is received. We handle this situation by discarding values received if there is no matching transaction. However, it may also be desirable to filter these transactions lower in the system, such that the committed transactions commit directly, and the squashed ones are cancelled in place. To control the size of this filter, we include a branch id with the transactions and track operations at the branch granularity. The filter can also be used to mark committed branches so that memory responses corresponding to them are committed directly.

Chapter 10

Leakage-Free Memory Hierarchy

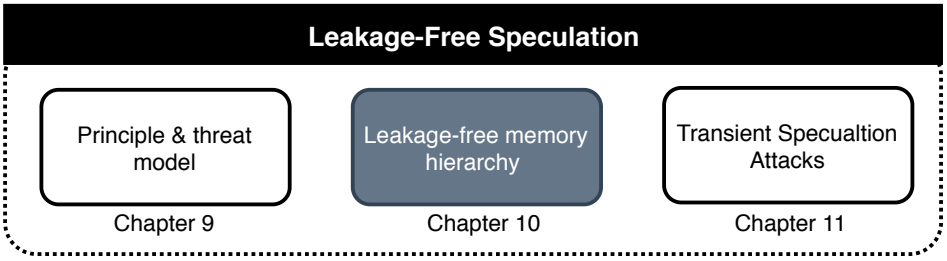


Figure 10.1: Leakage-free speculation project overview

After introducing the SafeSpec model to protect speculation by isolating speculative state from committed state (chapter 9). In this Chapter, we demonstrate the *SafeSpec* principle by building a memory hierarchy (caches and TLBs) that are free from speculation-induced leakage. Making the memory hierarchy speculation-leakage free prevents Meltdown and Spectre attacks. In particular, we expand the load-store queues to store a pointer to a temporary associative structure that holds speculatively loaded cache lines. We also introduce a similar structure to hold speculatively loaded translation lookaside buffer (TLB) entries. We describe the design and some of the complexity-performance trade-offs in Section 10.1. Furthermore, Section 10.2 presents a performance, complexity and security analysis of *SafeSpec*. We also analyze the complex-

ity of *SafeSpec* including the impact of all new structures and demonstrate a reasonable increase in the area and power consumption. Finally, we show that *SafeSpec* stops proof-of-concept implementations of all variants of Meltdown and Spectre, as well as the new variants that we introduced.

10.1 *SafeSpec* for Caches and TLBs

To demonstrate the *SafeSpec* principle, we implemented it to protect CPU caches and TLBs from leakage during speculative execution. To provide full protection, all speculatively updated structures should follow the SafeSpec principle. We chose the CPU caches because they are easily exploitable targets for covert communication and the ones used in the Spectre/Meltdown attacks; caches have simple indexing and with the availability of instructions such as `clflush` on x86, an attacker is able to evict data which facilitates quick exfiltration using, for example, a flush+reload covert channels [174].

To protect from speculative covert channels that occur during memory accesses, and following the *SafeSpec* principles, we need to add shadow state to protect the following structures (Figure 10.2).

Data caches: this is the covert channel used in all three Meltdown/Spectre variants. We add a shadow structure to hold the cache lines that have been fetched speculatively. The structure is associatively-filled lookup table (filled associatively, but accessed as a lookup-table). In the Load/Store queue, we point speculative loads that have received their data to a corresponding entry in this table. Speculative instructions in the *same execution branch as the load that fetched a shadow cache line* that accesses this cache line can use the value from the shadow structure. If an instruction commits (depending

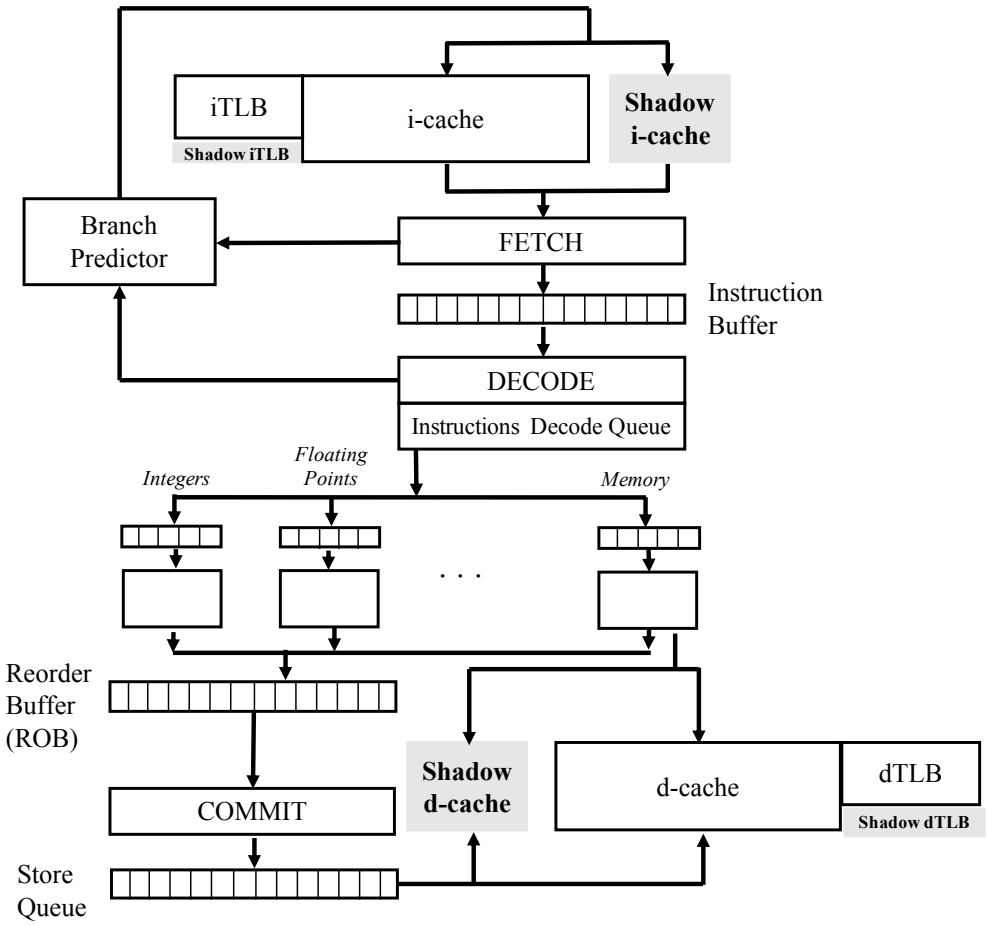


Figure 10.2: *SafeSpec* extension to the CPU pipeline to create leakage-free memory hierarchy (caches and TLBs).

on WFB or WFC), the cache line is moved from the shadow structure to the caches. If the instruction is squashed, the shadow structure entry is marked as available. Thus, not even the cache replacement algorithm state is affected by the speculative data that does not commit.

Instruction caches: we built variants of Meltdown/spectre using the instruction cache that replaces data dependent array access with dependent branches to a location in an array to disclose the data through the i-cache, illustrating that it must be protected as well 10.3. To develop this attack variant, we had to overcome branch predictor behavior: data dependent branches were using the branch predictor, rather than the secret data. Thus, we had to initialize the branch target buffer (BTB) to a third location, and then

introduce sufficient delay in the pipeline for the data dependent branch such that it has time to register the data dependent location in the i-cache.

TLBs: we also conjectured that the TLBs may be used as a covert channel vector. Given recent attacks such as Foreshadow [160] and TLBleed [51] which directly target the page translation behavior for speculation attacks, its critical to protect these structures. Essentially, the data dependent access would target a page based on the value of the data, causing the corresponding TLB entry to be initialized. Later, we can check the time to access the page to see if it results in a TLB miss or not to derive the communicated data.

To implement *SafeSpec* for the data cache, we add an associatively-filled lookup table to hold speculatively read cache lines. It is important to note that memory consistency models, such as Total Store Order (TSO) semantics of the x86-64, often ensure that store side-effects appear in order; in other words, the cache is not updated until the store commits, making stores robust to speculation attacks. We augment the load store queue with a pointer to the shadow cache line for load operations that are speculative. Any instruction dependent on the speculative load reads the cache line from the shadow structure. Once the load instruction commits, the shadow cache line is written to the caches according to the inclusion policy of the caches (in our case, since the caches are inclusive, it is written to all levels of the cache) and freed in the shadow structure. If the load is squashed, the value is freed in the shadow structure. For the i-cache and the TLBs, we create similar shadow structures, and augment the reorder buffer (ROB) with pointers to the shadow state entries if the instruction is speculative and the cache line (or TLB entry) were fetched speculatively.

From a performance perspective, the structures should be sized such that they accommodate the speculative state needed by representative workloads. If the shadow

```
attackMode <-- 0;
secret <-- readSecret();
int (*fnPtr)[256 * 256];
for all ascii in (ASCII-character)
{
    define int func_ascii() //noop sled
    {
        asm volatile(".rept 256;" "nop;"".endr;");
        return 0;
    }
    (*fnPtr)[ascii * 256] <-- func_ascii
}
clflush(&array1_size);
clflush(fnptr);
function speculative(secret)
{
    if (secret < array1_size)
        gadgetFunc(secret, attackMode);
}
function gadgetFunc(secret, attackMode)
{
    if (secret == 'A' && attackMode)
        fnPtr[A*256]();
    // ... 256 If Structures for
    // all ASCII characters
    if (secret == 'z' && attackMode)
        fnPtr[z * 256]();
junkLoc();

}
for (i = 1...256)
{
    t1 = rdtscp();
    junk = fnptr[i * 256](); //check cache hit
    t2 = rdtscp();
}
```

Figure 10.3: I-cache variant of Spectre

structures are full, we could either drop some of the shadow state (leading to loss of updates to the committed state with performance, rather than correctness implications), or block until there is space in the shadow state before issuing an instruction (also with performance implications). We will see later that the constraints introduced by security requirements to eliminate TSAs are more stringent than those required by performance. Figures 10.4 show the distribution of the size of the speculative state sampled over time for the SPEC 2017 benchmarks. The shadow d-cache for 3 of our benchmarks

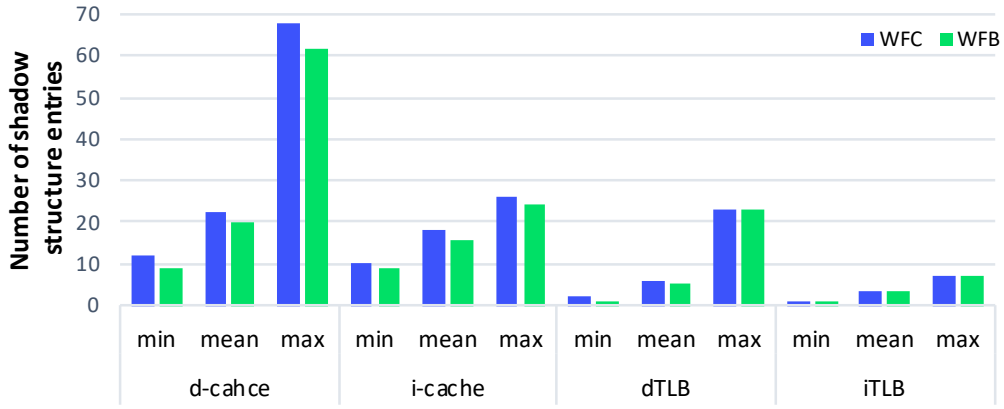


Figure 10.4: Shadow structure size that fits 99.99% of caches and TLBs accesses.

grows occasionally to almost the maximum possible size (bound by the size of the load-store queue). A shadow i-cache with about 25 cache lines is sufficient for all of the benchmarks. In addition, less than 10 entries are sufficient for speculative iTLB misses, but some benchmarks require more dTLB entries (up to 25). Given that the overhead of supporting WFC is small, we elect to support WFC to get the increased protection to cover Meltdown.

10.2 Evaluation

Table 10.1: Configuration of the Simulated architecture

Parameter	**Configuration**
CPU	6-way issue, 96 Issue Queue entries, out-of-order, no SMT, 72 Load Queue entries, 56 Store Queue entries, 224 ROB entries, 64 iTLB entries, 64 dTLB entries, commit up to 6 Micro-Ops/cycle
Private L1 i-/d-Cache	32 KB, 8-way, 64B line, 4 cycle hit
Shared L2 Cache	256 KB, 4-way, 64B line, 12 cycle hit
Shared L3 Cache	2 MB, 16-way, 64B line, 44 cycle hit

We conduct experiments with MARSSx86 [127], which is a cycle-accurate full-system simulator of out-of-order x86 cores. We configured the CPU and cache models of MARSSx86 to simulate the Intel Skylake processor as shown in Table 10.1.

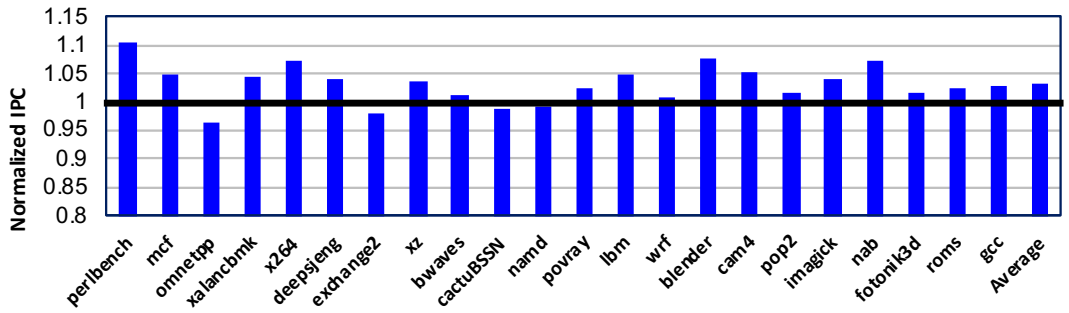


Figure 10.5: SafeSpec relative performance to non-secure OoO execution CPU using IPC values of running SPEC2017 benchmarks.

10.2.1 Performance Analysis

The first experiment measures the performance of compared to the baseline processor under conservative condition. In particular, we consider the shadow state access time to be equivalent to the access time of the L1 cache (4 cycles), when it is substantially smaller, and accessed as a lookup table. Figure 10.5, shows the IPC values for all SPEC2017 benchmarks. We see a small improvement in performance with a geometric mean of about 3%. We believe that this advantage results from a combination of effects including the larger effective cache size and avoiding polluting the cache with wrong path speculative state.

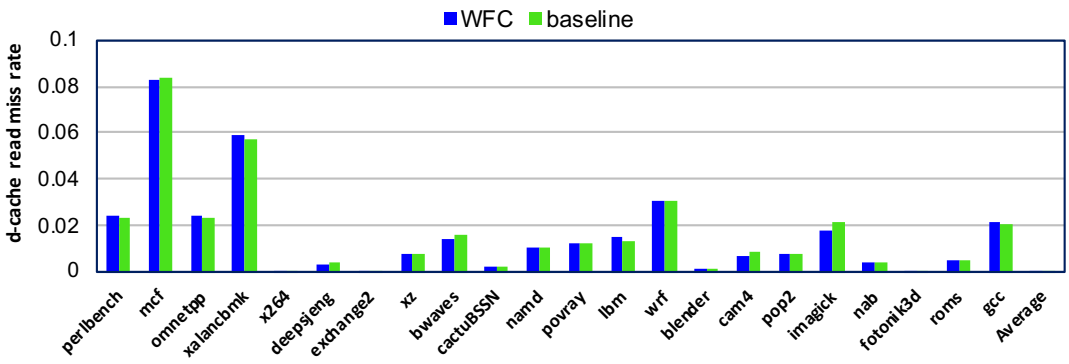


Figure 10.6: d-cache read miss rates including shadow d-cache

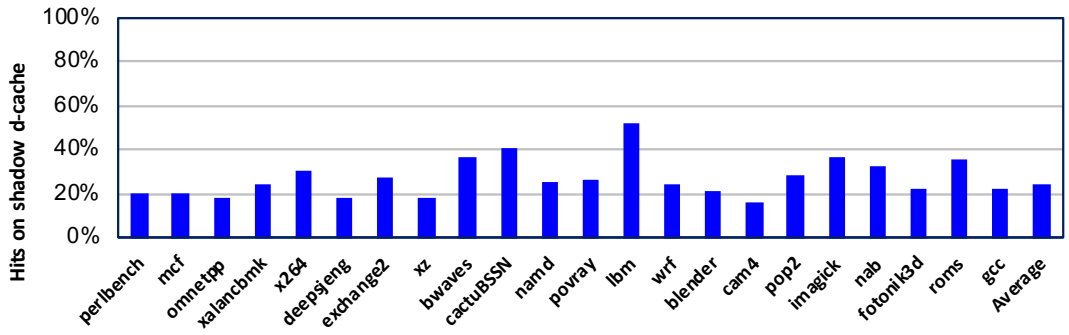


Figure 10.7: Percentage of hits on shadow d-cache

To gain more insight into the observed performance, Figure 10.6 shows the miss rate on read operations in the d-cache. There is little difference in behavior between SafeSpec and the baseline with respect to the data accesses. Figure 10.7 shows the percentage of the reads that hit the shadow structures.

The i-cache behavior is significantly different than the d-cache. Figure 10.8 shows the miss rate on the i-cache. For the i-cache, there are more substantial differences between WFC and the baseline. Some outlier behavior such as Pop2 and imagick where the percentage of i-cache misses drops significantly could be due to the larger size of the shadow structures expanding the effective size of the cache reducing conflict and capacity misses. Moreover, we see in Figure 10.9 that most of the hits occur in the shadow i-cache structure reflecting the high spatial locality of the access patterns in the i-cache; in other words, while a cache line is still speculative, several instructions execute from the same cache line. In contrast, the d-cache has less spatial locality, resulting in fewer accesses hitting the shadow state. We note that the cache miss rates are combined for all instructions (i.e., we do not exclude instructions that are squashed); therefore, many of these hits in the shadow structures may not end up being productive.

To understand the benefits of the shadow structure in filtering misspeculated accesses, Figure 10.10 shows the percentage of the shadow state that ends up being

committed for the i-cache and the d-cache. We observe that a substantially higher percentage of the d-cache state ends up being committed, perhaps due to the fact that speculative loads are issued later in the pipeline making them more likely to commit. For both the d-cache and especially the i-cache, the shadow structure filters a large number of misspeculated accesses that are squashed without cluttering the caches.

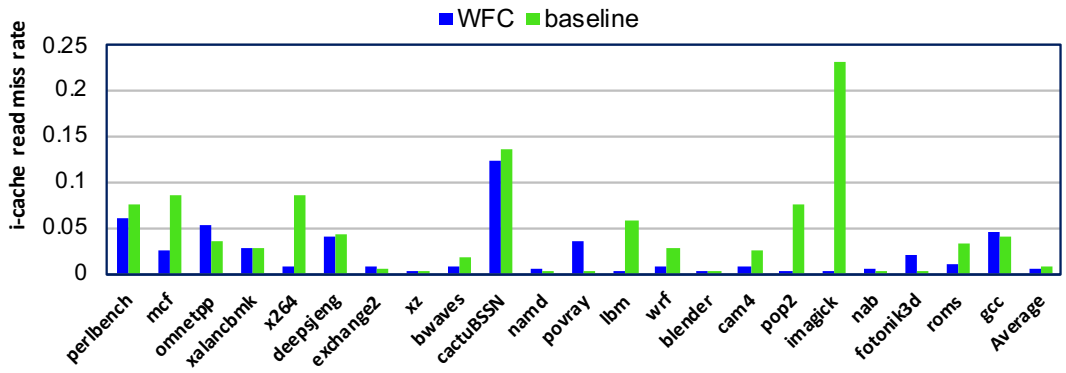


Figure 10.8: i-cache miss rate including the shadow i-cache

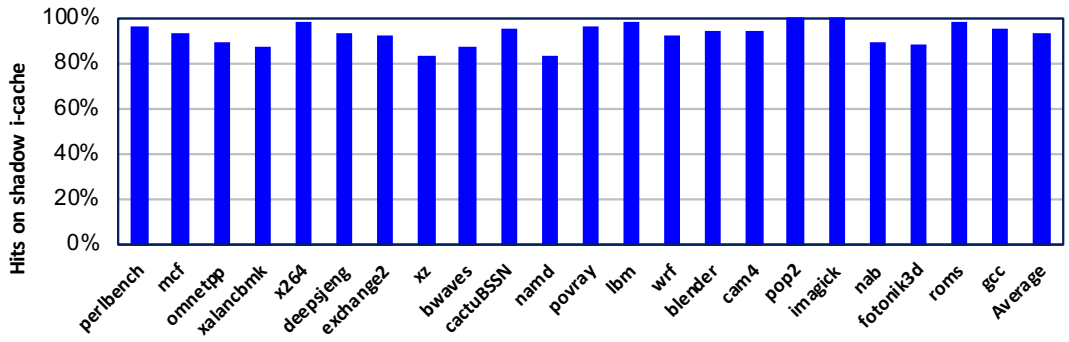


Figure 10.9: Percentage of hits on shadow i-cache

10.2.2 Security Analysis

Table 10.2 shows that both WFC and WFB close Spectre attacks, but only WFC is guaranteed to also stop Meltdown attacks. We evaluated our proof of concept code implementing Spectre in the simulator and found indeed that the attack fails under both WFC and WFB models. We evaluated the protection coverage for Spectre-style

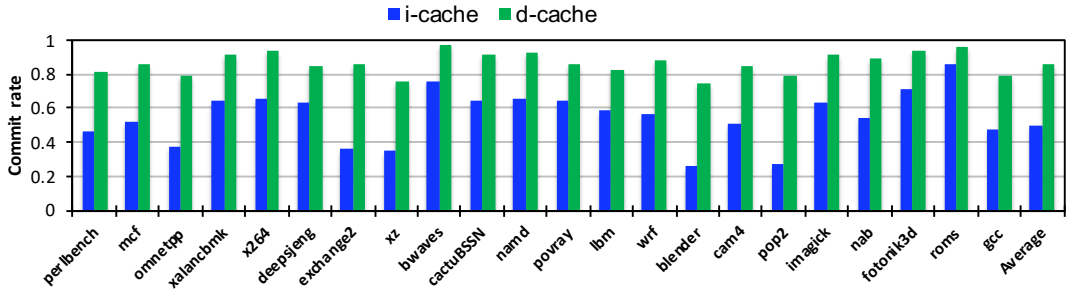


Figure 10.10: Commit rate of shadow state

Table 10.2: Security Analysis of Meltdown/Spectre

Spectre	**WFC**	**WFB**	**Meltdown**	**WFC**	**WFB**
Spectre-PHT [89, 86]	✓	✓	Meltdown [98]	✓	✗
Spectre-BTB [89]	✓	✓	Foreshadow [160, 167]	✓	✗
Spectre-STL [59]	✓	✗	Variant 3a [9]	✓	✗
Spectre-RSB [93, 105]	✓	✓	Lazy FP [151]	✓	✗
			Variant 1.2 [86]	✓	✗

attacks targeting structures other than the d-cache (i-cache, iTLB, and dTLB). All three side channels were closed. We tested proof of concept code for the i-cache and a transient attack through the d-cache and observed that the attack fails on the *SafeSpec* protected CPU. We could not get TLB-based attacks working in the simulator, perhaps because of the large delays of page walks, or due to the limitations of the MarSSx86 models of the TLBs.

10.2.3 Hardware overhead

Table 10.3: SafeSpec hardware overhead at 40nm.

	Power (mW)	**Power** (%)	**Area** (mm^2)	**Area** (%)
Secure	290.27	26.4	9.79	17
WFC	35.14	3	1.17	2

introduces hardware overheads to the CPU pipeline due to the addition of the shadow structures. We compared the hardware overhead for two different sizes for the shadow structures; 1) Secure: shadow structure size equal to the maximum

speculative state during speculation; and 2) SafeSpec with WFC: shadow structure sizes were optimized based on 99.99% speculative state size for SPEC2017 benchmarks using the WFC implementation. We report the area, power, and access time values, as well as a percentage compared to the Skylake CPU L1 cache configuration (shown in Table 10.1), using CACTI v5.3 [147] in Table 10.3. The results show that the area overhead is tolerable for the secure design, making the design highly practical.

Chapter 11

Transient side channel attacks

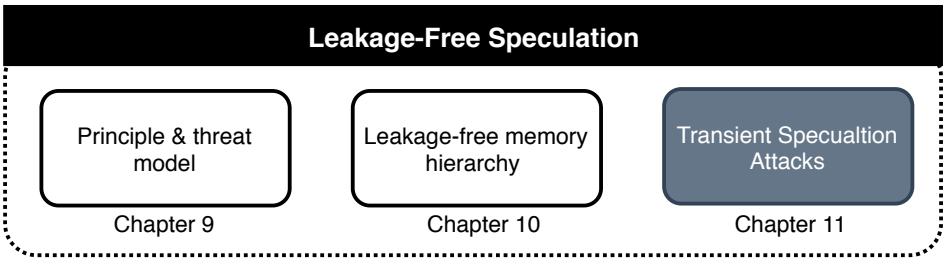


Figure 11.1: Leakage-free speculation project overview

After introducing the SafeSpec model to protect speculation by isolating speculative state from committed state (chapter 9) and demonstrating it by building a leakage-free memory hierarchy that prevents speculation attacks (Chapter 10). In this Chapter, we identify a transient type of leakage that occurs in the introduced speculative state (byproduct of *SafeSpec*) that we call *transient speculation attacks* (TSAs); *SafeSpec* by construction prevents speculative values from affecting the state of committed structures, which is the pathway used to communicate data covertly in the published speculation attacks. However, it does not create isolation between instructions that are in the speculative state. This creates a possibility for a new variant of attacks which we call *transient speculation attacks (TSAs)*. In particular, since most instructions that

commit start in the speculative state, there is a window of time where they can share the speculative state with misspeculated instructions before they are squashed. If we are not careful, it is possible to create a covert channel in this period to communicate the sensitive data from the mis-speculated branch to the branch that will be committed, allowing the data to be exfiltrated.

Consider the example of a shadow structure that is sized to be small (let's say one entry). The malicious speculative code that reads the privileged data can then communicate it covertly to speculative code (the "receiver" code that will commit) using the shadow state. For example, it can replace the entry in the shadow state, causing the receiver to notice the absence of its speculative state (since it was replaced) after it commits. Alternatively, if we block when the shadow structure is full, the receiver can detect that its code took a longer time to execute.

Although TSAs are strictly less powerful than the original attack, they must be carefully considered to ensure that leakage is not possible. One way to solve this problem is to either partition the speculative state per branch, or to size it generously, or even for the worst case scenario, to ensure that no leakage occurs through the shadow state. TSAs can also attempt to communicate covertly by creating contention on functional units or other shared structures; this is an issue that we also consider. We discuss how to mitigate TSA attacks in Section 11.1.

11.1 Transient Speculation Attacks: Covert Channels in the Speculative State

The *SafeSpec* principle prevents direct side-channel leakage from the speculative state to the committed state, closing all known speculation attacks. However,

although the committed instructions and the speculative instructions eventually reside in separate structures, creating the separation and closing the channel, eventually committed instructions can start out as speculative. During this window, the eventually committed instructions share the shadow state with any speculative instructions that will be squashed. If the shadow structures are not designed carefully, covert channels can be created during this transient window to communicate sensitive data (which can only be read by a mis-speculated path) to an instruction pathway that will be committed such that the leakage results are visible to the program. It is important to emphasize that these attacks (which we call Transient Speculation Attacks, or TSAs) are substantially more difficult than Spectre/Meltdown because there is only a limited window of speculation in which the malicious Trojan code must not only read sensitive data, but also create measurable contention to the spy before either of their predicate branches commits.

TSAs are possible only if the shadow structures are shared and sized such that they enable contention. Consider an example where we size the TLB shadow structures based on typical program behavior. Since programs do not have many pending TLB misses within a speculation window, it stands to reason to size these structures to be small. In the rare case when the shadow structures are full, we may handle this by either discarding updates or by blocking the issue of requests when there is no room in the shadow structure. Either of these behaviors provides potential for a covert channel. Consider that the Trojan fills the structures with TLB misses if it wants to communicate a 1. If updates are discarded, a spy can detect a communication if its TLB accesses are not committed (they were discarded). Alternatively, if we block TLB accesses when the structures are full, the spy can detect a communication of 1 if its TLB accesses are delayed causing a longer TLB miss time. The attack is illustrated in Figure 11.2.

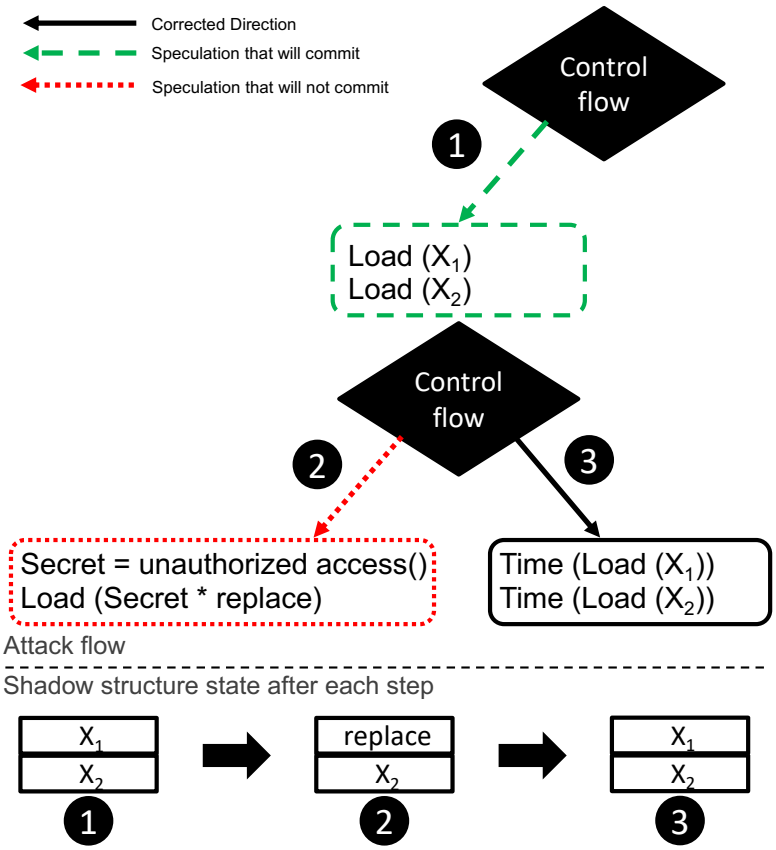


Figure 11.2: Transient speculation attack (TSA) overview

To prevent TSAs through the shadow structures, we elect to provision them for the worst case scenario to make sure that transient contention cannot be created within a speculation window. This approach guarantees that no contention on the shadow structures is possible, at the cost of provisioning fairly large associative structures. We believe that with some more analysis, or with some detection defense that detects an attack when the shadow structures grow abnormally large, this worst case provisioning can be substantially relaxed without introducing leakage.

Chapter 12

Conclusions and Future Work

The increasing number of attacks on computing systems motivated the need to consider cybersecurity as a fundamental requirement for designing and evaluating computing systems on par with performance, functionality, or power consumption. Furthermore, the limitations of software solutions necessitate the need to view security as hardware/software problem – can originate in hardware/software and to be solved by hardware and/or software. Therefore, in this dissertation, we pursue approaches to building defenses in the architecture to help make the software more secure. Specifically, introduced EnsembleHMDs that improve the performance of single HMDs using an ensemble of specialized detectors. Furthermore, we showed that current implementations of HMDs are not resilience to evasion attacks and proposed a new constriction for HMDs (called RHMDs) to make HMDs more resilience to evasive malware. Finally, we proposed a principled architecture solution to deafened against speculation attacks with minimum impact on performance. Although we addressed the limitations/threat models of the problems proposed in this dissertation, these areas still have open research problems. In this chapter, we are going to discuss completed research and future work.

12.1 Accurate Hardware Malware Detectors

In this project, we seek to improve the detection performance of hardware malware detectors (HMDs) through ensemble learning to increase the efficiency of a Two-Level Detector (TLD). We envision an HMD that uses low level features to provide the first line of defense to detect suspicious processes. This detector then prioritizes the effort of a heavyweight software detector to look only at programs that are deemed suspicious, forming a TLD.

We started by evaluating whether specialized detectors can be more effectively classify one given class of malware. We found out that this is almost always true for the features and malware types we considered. We then examined different ways of combining general and specialized detectors. We found that ensemble learning by combining general detectors provided a limited advantage over a single general detector. However, combining specialized detectors can significantly improve the sensitivity, specificity, and accuracy of the detector.

We developed metrics to evaluate the performance advantage from better detection in the context of a TLD. Ensemble learning provides more than 16.6x reduction in the online detection overhead with the NN specialized ensemble detector. This represents 2x improvement in performance (overhead) with respect to Ozsoy et al. [123] single detector implementation. We implemented the proposed detector as part of an open core to study the hardware overhead. The hardware overhead was minimal: around 2.88% increase in area, 9.83% reduction in cycle time, and less than 1.35% increase in power. We believe that minor optimization of the MEM feature collection circuitry could alleviate most of the cycle time reduction.

We compared the performance and overhead of LR to NN as base classifiers. NN based ensemble detectors provide the highest classification accuracy of all the detectors we developed. Although they are more complicated to implement, we used hardware optimizations to reuse a single perceptron sequentially to implement the neural network, making the hardware overhead small compared to LR.

Finally, we carried out a study of how the detector performs as malware evolves over time. Specifically, we trained a detector with an old malware data set and evaluated its performance on both malware from the same generation, as well as more recent malware. We discovered that the detection performance rapidly deteriorates as malware evolves. In addition, the detectors also fail when trained using only recent malware when classifying old malware. These results emphasize the necessity to provide a secure way to update the hardware detector weights and thresholds as malware continues to evolve.

For our future work, we will study the effect of choosing a threshold for the detectors that favors sensitivity or specificity over the other, by inserting weights for them in the objective function, on the performance of the two-level detection system.

12.2 Evasion-Resilient Hardware Malware Detectors

In this project, we targeted the recently proposed Hardware Malware Detectors which have demonstrated remarkable accuracy in classifying malware using low-level features. In one model, when implemented in hardware, they can help monitor all programs as they run to detect malware with high accuracy, and at a cost order of magnitude lower than software monitoring. There is evidence that these detectors will be incorporated in commercial processors [132].

If HMDs are widely deployed, we must expect that attackers will attempt to evade detection as is the case in any adversarial setting. We believe that this is the first paper that considers this issue in detail for HMDs. In particular, we showed that the attacker can accurately reverse-engineer earlier proposed detectors. Moreover, once the detector is reverse-engineered, we demonstrated simple evasive techniques that can successfully hide malware from detection. Although existing HMDs admit that these detectors may be vulnerable to evasion, these results conclusively confirm that malware can evade such detectors, rendering them ineffective.

We considered that detectors can be retrained to capture evasive malware once samples become known (similar to the current practice of updating virus signatures). For LR, such retraining was not effective: considering evasive malware compromised the classification performance on normal malware; due to linear separation, it is not possible to produce LR detectors that successfully catch both. In contrast, NN could easily be retrained to detect evasive malware. Thus, non-linear classifiers need to allow the detectors to be retrained to capture emerging malware. However, after several rounds of evade-retrain game, the NN classifier could also no longer be effectively retrained.

We proposed resilient HMDs that switch randomly between a diversity of detectors. We showed that such detectors can resist reverse-engineering and complicate evasion. We showed both empirically, and from PAC learnability theory, that their resilience increases with the number and diversity of the individual detectors available to select from. With this class of resilient HMDs, hardware malware detection becomes a promising direction in malware detection.

In our future work, we will investigate more powerful attacks on RHMDs; although RHMDs are resilient to reverse engineering since they are unpredictable (the output may differ for the same input when querying the victim multiple times), a more

powerful attacker could use this feature to her advantage to create a new label that can be used to train the reverse-engineered detector. For example, consider the following possible attack. The attacker would first query the victim RHMD using the attacker training data set multiple times (enough to query all base detectors in the RHMDs). As a results, for each input from the attacker training data set, three output scenarios may happen: (1) Malware (M): all base detectors agree that the input is a malware, (2) Regular (R): all base detectors agree that the input is a regular program, and (3) Unknown (X): some base detectors think this programs in a malware while the others think that it is a regular program. Figure 12.1, shows an overview of possible outputs of an RHMD that have two base detectors. Therefore, if the attacker labeled the unpredictable input with a different label than malware or regular program (Unknown (X) in our example), then train the reverse engineered detector on the labeled data, the reverse-engineered detector would have a model that can distinguish between the three classes that the attacker trained the model on. Now, since the attacker has access to the reverse-engineered model, the attacker can inject instructions in a way that moves the decision of RHMD from being malware across all decision boundaries (of all base detectors) to being regular.

Furthermore, to defend against such powerful attacks, we would like to investigate the creation of more powerful RHMDs. In particular, we want to build RHMDs with base detectors that have the following features:

- Large number of base detectors: by increasing the number of base detectors we are increasing the number of quires the attacker needs to do to query all base-detectors, which is needed for the more powerful attack. In addition, by figuring out fastest time the attacker can query all base detectors which should be very long

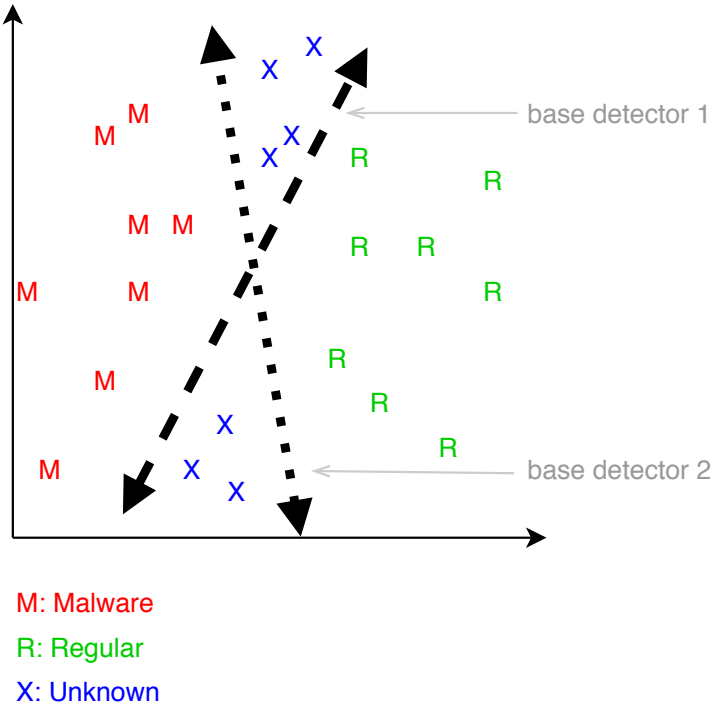


Figure 12.1: Overview of RHMD possible outputs given 2 base detectors

(for example, days, weeks, months, or even years), this can be set as a threshold to update the base detectors so that the attack would fail. It would also be interesting to come up with automatic update strategies for the base detectors in a random way.

- Different detection spaces: use different feature spaces for different base detectors so that if an evasive sample is created to evade detection in one feature space does not necessarily evade detection in other spaces. For example, changing the instructions that the malware is using to create evasive sample does not change the malware memory accesses footprint.

- Zero transferability between base detectors: transferability between two detectors (A and B), is the percentage of evasive samples that are crafted to evade detector A that can also evade detector B. Therefore, if we can create base detectors that

have zero transferability between them, then crafting evasive samples that can evade detection of all base detectors would be very hard.

Essentially, these techniques make the composition space of HMDs large and time-variant, making it difficult to construct attacks that evade them. Selection of low transferability detectors also complicates the development of successful attacks against reverse engineered models. We leave these advanced attacks and the detailed exploration of such defenses to future work.

12.3 Leakage Free Speculation

We presented a general principle for supporting speculative execution in a way that makes out-of-order processors immune to speculation-based attacks. The principle relies on leaving speculative state in shadow structures, and only committing this state once the instructions that generate them are guaranteed to commit. Thus, side-effects of misspeculation are hidden from the primary structures of the CPU, closing the vulnerability. We demonstrated the principle of protecting caches and TLBs of the CPU. Our design closes new variants of attacks that we developed to leak through the i-cache or the TLBs. We showed that careful design is needed to prevent a form of leakage that can arise while instructions share the speculative state. We mitigate this leakage by sizing the speculative state conservatively. Constructed this way, transient attacks also become impractical. The performance of the *SafeSpec* CPU was actually slightly higher than an unmodified CPU, despite conservative estimates on the shadow state. We believe that the presented design represents a first step of many towards a principled protection of speculative execution. To provide complete protection, other microarchitectural states that can be updated speculatively should use the same principle.

One of *SafeSpec*'s limitations is that it requires a deep redesign of the CPU to separate out the speculative state from the permanent state. It also has implications on security: we identified a form of transient side channels that occur through the shadow structures. The goal of this paper is to establish the *SafeSpec* principle by protecting the CPU caches and TLBs. We recognize that other structures affected by speculative instructions must also be protected using this principle or otherwise the attackers will switch to using them. Future work should look at protecting the branch predictor, DRAM buffers, account for prefetchers, as well as other structures.

Another limitation of *SafeSpec* is that we do not support multi-threaded workloads. Addressing this limitation involves two considerations. The more straightforward consideration is how to preserve the semantics of protocols such as cache coherence, memory consistency models, atomic operations, and transactional memory. We believe that these continue to operate in the same way by treating the speculative state to be part of the state of the caches. The second issue is significantly more difficult: these protocols themselves can be used to communicate speculative side-effects as has been recently shown by the MeltdownPrime attack [157]. Designing leakage-free protocols is a difficult problem that deserves separate and complete treatment and therefore we elected to leave supporting multi-threaded workloads to future work.

Bibliography

[1] Altera de2-115 development and education board. https://www.altera.com/solutions/partners/partner-profile/terasic-inc-/board/altera-de2-115-development-and-education-board.html#overview, 2010.

[2] ADVANCED MICRO DEVICES, I. Software techniques for managing speculation on amd processors. https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018.

[3] Aldrich, C., and Auret, L. *Unsupervised process monitoring and fault diagnosis with machine learning methods*. Springer, 2013.

[4] Aleksander, O. The ao486 project. https://github.com/alfikpl/ao486, 2014.

[5] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. Are your training datasets yet relevant? In *International Symposium on Engineering Secure Software and Systems* (2015), Springer, pp. 51–67.

[6] Arends, J., and Kerstin, I. Malware analysis.

[7] Arlot, S., Celisse, A., et al. A survey of cross-validation procedures for model selection. *Statistics surveys 4* (2010), 40–79.

[8] ARM. Cache speculative side-channels. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[9] ARM. Vulnerability of speculative processors to cache timing side-channel mechanism. https://developer.arm.com/support/security-update, 2018.

[10] Aung, Z., and Zaw, W. Permission-based android malware detection. *International Journal of Scientific and Technology Research 2*, 3 (2013), 228–234.

[11] Malware Statistics, 2019. Available online: http://www.av-test.org/en/statistics/malware/.

[12] Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. The security of machine learning. *Machine Learning 81*, 2 (2010), 121–148.

[13] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), vol. 41, p. 46.

[14] Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. Smotherspectre: exploiting speculative execution through port contention. *arXiv preprint arXiv:1903.01843* (2019).

[15] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2013), Springer, pp. 387–402.

[16] Biggio, B., Fumera, G., and Roli, F. Adversarial pattern classification using multiple classifiers and randomisation. *Structural, Syntactic, and Statistical Pattern Recognition* (2008), 500–509.

[17] Biggio, B., Nelson, B., and Laskov, P. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).

[18] Bilar, D. Opcode as predictor for malware. *International Journal of Electronic Security and Digital Forensic* (2007).

[19] Bloch, E. The engineering design of the stretch computer. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference* (1959).

[20] Bruening, D. L. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[21] Bruschi, D., Cavallaro, L., and Lanzi, A. An efficient technique for preventing mimicry and impossible paths execution attacks. In *Proc. of Performamce, Computing and Communications Conference (IPCCC)* (2007).

[22] Caballero, J., Grier, C., Kreibich, C., and Paxson, V. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium* (2011).

[23] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441* (2018).

[24] Carruth, C. Mitigating speculative attacks in crypto. `https://github.com/HACS-workshop/spectre-mitigations/blob/master/crypto_guidelines.md`, 2018.

[25] Carruth, C. Rfc: Speculative load hardening (a spectre variant 1 mitigation). `https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html`, 2018.

[26] Chatterjee, R., Doerfler, P., Orgad, H., Havron, S., Palmer, J., Freed, D., Levy, K., Dell, N., McCoy, D., and Ristenpart, T. The spyware used in intimate partner violence. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 441–458.

[27] Chavis, E., Davis, H., Hou, Y., Hicks, M., Yitbarek, S. F., Austin, T., and Bertacco, V. SNIFFER: A high-accuracy malware detector for enterprise-based systems. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)* (July 2017), pp. 70–75.

[28] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proc. of CCS* (2010), ACM.

[29] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[30] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).

[31] Chien, E. Techniques of adware and spyware. In *the Proceedings of the Fifteenth Virus Bulletin Conference, Dublin Ireland* (2005), vol. 47.

[32] Christodorescu, M., Jha, S., Seshia, S., Song, D., and Bryant, R. Semantics-aware malware detection. In *Proc. IEEE Symposium on Security and Privacy* (2005), pp. 32–46.

[33] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E. Semantics-aware malware detection. In *Proc. IEEE Symposium on Security and Privacy (SP)* (2005), pp. 32–46.

[34] C.Luk, Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI* (2005).

[35] Colbaugh, R., and Glass, K. Predictive defense against evolving adversaries. In *Intelligence and Security Informatics (ISI), 2012 IEEE International Conference on* (2012), IEEE, pp. 18–23.

[36] Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of 40th IEEE Symposium on Security and Privacy (S&P19)* (2019).

[37] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. On the feasibility of online malware detection with performance counters. In *Proc. Int. Symposium on Computer Architecture (ISCA)* (2013).

[38] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News 41*, 3 (2013), 559–570.

[39] Dietterich, T. G. Machine learning research: Four current directions, 1997.

[40] Dietterich, T. G. Ensemble methods in machine learning. In *International workshop on multiple classifier systems* (2000), Springer, pp. 1–15.

[41] Dillon, M. Clarifying the spectre mitigations. `http://lists.dragonflybsd.org/pipermail/users/2018-January/335637.html`, 2018.

[42] Drucker, H., Wu, D., and Vapnik, V. N. Support vector machines for spam categorization. *IEEE Transactions on Neural networks 10*, 5 (1999), 1048–1054.

[43] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys 44*, 2 (Mar. 2008).

[44] Equifax. Equifaxs statement for the record regarding the extent of the cybersecurity incident announced on september 7, 2017, 2017. Available online at: `https://www.sec.gov/Archives/edgar/data/33185/000119312518154706/d583804dex991.htm`.

[45] Eskandari, M., and Hashemi, S. Metamorphic malware detection using control flow graph mining. *International Journal of Computer Science and Network Security 11*, 12 (2011), 1–6.

[46] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump over ASLR: Attackiung branch predictors to bypass aslr. In *Proc. IEEE International Symposium on Microarchitecture (MICRO)* (2016).

[47] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO) 13*, 1 (2016), 10.

[48] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N., and Ponomarev, D. Branchscope: A new side-channel attack on directional branch predictor. In *Proc. of ASPLOS* (2018).

[49] Folino, G., Pizzuti, C., and Spezzano, G. Gp ensemble for distributed intrusion detection systems. In *Pattern Recognition and Data Mining*. 2005, pp. 54–62.

[50] Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[51] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proc. of USENIX Security* (2018).

[52] Gregg, B. KPTI meltdown initial performance regressions, 2018. `http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html`.

[53] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is dead: long live KASLR. In *Proc. of ESSoS* (2017).

[54] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+ flush: a fast and stealthy cache attack. In *Proc. of DIMVA* (2016).

[55] Hawkins, D. M. The problem of overfitting. *Journal of chemical information and computer sciences 44*, 1 (2004), 1–12.

[56] Henning, J. L. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[57] Hofmeyr, S., Forrest, S., and Somayaji, A. Intrusion detection using sequences of system calls. *Journal of computer security 6*, 3 (1998), 151–180.

[58] Horn, J. Reading privileged memory with a side-channel, 2018.

[59] Horn, J. speculative execution, variant 4: speculative store by-pass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018.

[60] Hosmer Jr., D. W., and Lemeshow, S. *Applied Logistic Regression*. John Wiley & Sons, 2004.

[61] Hou, S., Chen, L., Tas, E., Demihovskiy, I., and Ye, Y. Cluster-oriented ensemble classifiers for intelligent malware detection. In *Semantic Computing (ICSC), 2015 IEEE International Conference on* (2015), IEEE, pp. 189–196.

[62] Huang, D. Y., Aliapoulios, M. M., Li, V. G., Invernizzi, L., Bursztein, E., McRoberts, K., Levin, J., Levchenko, K., Snoeren, A. C., and McCoy, D. Tracking ransomware end-to-end. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 618–631.

[63] Hunger, C., Kazdagli, M., Rawat, A., Dimakis, A., Vishwanath, S., and Tiwari, M. Understanding contention-based channels and using them for defense. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 639–650.

[64] Idika, N., and Mathur, A. A survey of malware detection techniques. Technical Report, Departemnt of Computer Science, Purdue University. Accessed Feb. 2014 at: `http://cyberunited.com/wp-content/uploads/2013/03/A-Survey-of-Malware-Detection-Techniques.pdf`.

[65] Inc., O. S. S. Respectre: The state of the art in spectre defenses. `https://www.grsecurity.net/respectre_announce.php`, 2018.

[66] Inc, U. Losing face: Two more cases of third-party facebook app data exposure, April 2019. Available online at: `https://www.upguard.com/breaches/facebook-user-data-leak`.

[67] Intel. Intel analysis of speculative execution side channels. `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf`, 2018.

[68] Intel. Retpoline: A branch target injection mitigation. `https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf`, 2018.

[69] Intel. Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, 2018.

[70] Intel. Speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[71] Jacob, G., Debar, H., and Filiol, E. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology 4*, 3 (2008), 251–266.

[72] Jiménez, D. A., and Lin, C. Dynamic branch prediction with perceptrons. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)* (2001), pp. 197–206.

[73] Kayaalp, M., Khasawneh, K. N., Esfeden, H. A., Elwell, J., Abu-Ghazaleh, N., Ponomarev, D., and Jaleel, A. Ric: relaxed inclusion caches for mitigating llc side-channel attacks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2017), IEEE, pp. 1–6.

[74] Kayaalp, M., Nomani, J., Schmitt, T., Ponomarev, D., and Abu-Ghazaleh, N. Scrap: Architecture for signature-based protection from code reuse attacks. In *International Symposium on High Performance Computer Architecture (HPCA)* (Feb. 2013).

[75] Kayaalp, M., Ponomarev, D., Abu-Ghazaleh, N., and Jaleel, A. A high-resolution side-channel attack on last-level cache. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE* (2016), IEEE, pp. 1–6.

[76] Kazdagli, M., Huang, L., Reddi, V., and Tiwari, M. EMMA: A new platform to evaluate hardware-based mobile malware analyses. *CoRR abs/1603.03086* (2016).

[77] Kazdagli, M., Reddi, V. J., and Tiwari, M. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* (2016), IEEE, pp. 1–13.

[78] Khasawneh, K. *Ensemble Learning with an Architectural Sub-semantic Engine for Malware Detection*. PhD thesis, State University of New York at Binghamton, 2014.

[79] Khasawneh, K. N., Abu-Ghazaleh, N., Ponomarev, D., and Yu, L. RHMD: Evasion-resilient hardware malware detectors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, ACM, pp. 315–327.

[80] Khasawneh, K. N., Abu-Ghazaleh, N. B., Ponomarev, D., and Yu, L. Adversarial evasion-resilient hardware malware detectors. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2018), IEEE, pp. 1–6.

[81] Khasawneh, K. N., Koruyeh, E. M., Song, C., Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *arXiv preprint arXiv:1806.05179* (2018).

[82] Khasawneh, K. N., Ozsoy, M., Donovick, C., Abu-Ghazaleh, N., and Ponomarev, D. Ensemble learning for low-level hardware-supported malware detection. In *International Workshop on Recent Advances in Intrusion Detection* (2015), Springer, pp. 3–25.

[83] Khasawneh, K. N., Ozsoy, M., Donovick, C., Abu-Ghazaleh, N., and Ponomarev, D. Ensemble learning for low-level hardware-supported malware detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404* (New York, NY, USA, 2015), RAID 2015, Springer-Verlag New York, Inc., pp. 3–25.

[84] Khasawneh, K. N., Ozsoy, M., Donovick, C., Ghazaleh, N. A., and Ponomarev, D. V. Ensemblehmd: Accurate hardware malware detectors with specialized ensemble classifiers. *IEEE Transactions on Dependable and Secure Computing* (2018).

[85] Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., and Emer, J. Dawg: A defense against cache timing attacks in speculative execution processors.

[86] Kiriansky, V., and Waldspurger, C. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[87] Kloft, M., and Laskov, P. Online anomaly detection under adversarial impact. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (2010), pp. 405–412.

[88] Kocher, P. Spectre mitigations in microsoft's c/c++ compiler. `MicrosoftCompilerSpectreMitigation.html`, 2018.

[89] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *Proc. of S&P* (2019).

[90] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (Oakland)* (2019).

[91] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. Effective and efficient malware detection at the end host. In *USENIX Security Symposium* (2009), pp. 351–366.

[92] Kolter, J. Z., and Maloof, M. A. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research 7* (2006), 2721–2744.

[93] Koruyeh, E., Khasawneh, K., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *Proc. of WOOT* (2018).

[94] Kruegel, C., Robertson, W., and Vigna, G. Detecting kernel-level rootkits through binary analysis. In *Proc. Annual Computer Security Applications Conference (ACSAC)* (2004), pp. 91–100.

[95] Kuon, I., and Rose, J. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems 26*, 2 (2007), 203–215.

[96] Laskov, P., and Šrndić, N. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 197–211.

[97] Lee, M. Malware detectors learn to work together. *Nature Electronics 1*, 4 (2018), 206.

[98] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *Proc. of USENIX Security* (2018).

[99] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium (Security)* (2018).

[100] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 605–622.

[101] Liu, J.-C., Song, J.-F., Miao, Q.-G., Cao, Y., and Quan, Y.-N. An ensemble cost-sensitive one-class learning framework for malware detection. *International Journal of Pattern Recognition and Artificial Intelligence* (2012), 1550018.

[102] Lord, B. An important message about yahoo user security, September 2016. Available online at: `https://yahoo.tumblr.com/post/150781911849/an-important-message-about-yahoo-user-security`.

[103] Lu, Y.-B., Din, S.-C., Zheng, C.-F., and Gao, B.-J. Using multi-feature and classifier ensembles to improve malware detection. *Journal of CCIT 39*, 2 (2010), 57–72.

[104] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005).

[105] Maisuradze, G., and Rossow, C. ret2spec: Speculative execution using return stack buffers. In *Proc. of CCS* (2018).

[106] Maisuradze, G., and Rossow, C. Speculose: Analyzing the security implications of speculative execution in CPUs. *arXiv preprint arXiv:1801.04084* (2018).

[107] Martignoni, L., Christodorescu, M., and Jha, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *IEEE Annual Computer Security Applications Conference (ACSAC)* (2007), pp. 431–441.

[108] Microsoft. Spectre mitigations in msvc. `https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/`, 2018.

[109] How Microsoft antimalware products identify malware and unwanted software, 2017. Available online: `https://www.microsoft.com/security/portal/mmpc/shared/objectivecriteria.aspx`.

[110] Minkin, M., Moghimi, D., Lipp, M., Schwarz, M., Van Bulck, J., Genkin, D., Gruss, D., Sunar, B., Piessens, F., and Yarom, Y. Fallout: Reading kernel writes from user space.

[111] Moser, A., c. Kruegel, and Kirda, E. Limits of static analysis of malware detection. In *IEEE Annual Computer Security Applications Conference (ACSAC)* (2007), pp. 421–430.

[112] Mutlu, O., Patt, Y. N., Kim, H., and Armstrong, D. N. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers 54* (2005), 1556–1571.

[113] Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. N. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on* (2003), IEEE, pp. 129–140.

[114] Nachenberg, C. Computer virus-antivirus coevolution. *Communications of the ACM 40*, 1 (Jan. 1997), 46–51.

[115] Naghibijouybari, H., and Abu-Ghazaleh, N. Covert channels on gpgpus. *IEEE Computer Architecture Letters 16*, 1 (2017), 22–25.

[116] Natani, P., and Vidyarthi, D. Malware detection using API function frequency with ensemble based classifier. In *Security in Computing and Communications*. Springer, 2013, pp. 378–388.

[117] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (2007).

[118] Malwaredb Website, 2015. Available online (last accessed, May 2015): `www.malwaredb.malekal.com`.

[119] Nohe, P. Autopsying the marriott data breach: This is why insurance matters, March 2019. Available online at: `https://www.thesslstore.com/blog/autopsying-the-marriott-data-breach-this-is-why-insurance-matters/`.

[120] Oleksenko, O., Trach, B., Reiher, T., Silberstein, M., and Fetzer, C. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506* (2018).

[121] Ozdemir, M., and Sogukpinar, I. An android malware detection architecture based on ensemble learning. *Transactions on Machine Learning and Artificial Intelligence 2*, 3 (2014), 90–106.

[122] Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. Malware-aware processors: A framework for efficient online malware detection. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 651–661.

[123] Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. Malware aware processors: A framework for efficient online malware detection. In *Proc. Int. Symposium on High Performance Computer Architecture (HPCA)* (2015).

[124] Ozsoy, M., Khasawneh, K. N., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers 65*, 11 (2016), 3332–3344.

[125] Ozsoy, M., Khasawneh, K. N., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers 65*, 11 (2016), 3332–3344.

[126] Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697* (2016).

[127] Patel, A., Afram, F., and Ghose, K. Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors. In *Proc. of QUF* (2011).

[128] Peddabachigari, S., Abraham, A., Grosan, C., and Thomas, J. Modeling intrusion detection system using hybrid intelligent systems. *Journal of network and computer applications 30*, 1 (2007), 114–132.

[129] Percival, C. Cache missing for fun and profit, 2005.

[130] Poushter, J., and Fetterolf, J. International publics brace for cyberattacks on elections, infrastructure, national security, January 2019. Available online at: `https://www.pewresearch.org/global/2019/01/09/international-publics-brace-for-cyberattacks-on-elections-infrastructure-national-security/`.

[131] Projects, T. C. Site isolation. `http://www.chromium.org/Home/chromium-security/site-isolation`.

[132] Qualcomm smart protect technology, 2016. Last Accessed July 2016 from `https://www.qualcomm.com/products/snapdragon/security/smart-protect`.

[133] Qualcomm smart protect technology, 2016. Last Accessed September 2016 from `https://www.qualcomm.com/products/snapdragon/security/smart-protect`.

[134] Quinlan, J. R. Simplifying decision trees. *Int. J. Man-Mach. Stud. 27*, 3 (Sept. 1987), 221–234.

[135] Roesch, M. Snort: Lightweight intrusion detection for networks. In *Proc. Usenix System Adminsitration Conference (LISA)* (1999), pp. 229–238.

[136] Runwal, N., Low, R. M., and Stamp, M. Opcode graph similarity and metamorphic detection. *J. Comput. Virol. 8*, 1-2 (May 2012), 37–52.

[137] Salamatian, S., Huleihel, W., Beirami, A., Cohen, A., and Médard, M. Why botnets work: Distributed brute-force attacks need no synchronization. *IEEE Transactions on Information Forensics and Security* (2019).

[138] Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., and Hamze, A. Malware detection based on mining API calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), SAC '10, ACM, pp. 1020–1025.

[139] Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C., and Bringas, P. G. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems*. Springer, 2010, pp. 35–43.

[140] Schwarz, M., Canella, C., Giner, L., and Gruss, D. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. *arXiv preprint arXiv:1905.05725* (2019).

[141] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. Zombieload: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726* (2019).

[142] Schwarz, M., Schilling, R., Kargl, F., Lipp, M., Canella, C., and Gruss, D. Context: Leakage-free transient execution. *arXiv preprint arXiv:1905.09100* (2019).

[143] Schwarz, M., Schwarzl, M., Lipp, M., and Gruss, D. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535* (2018).

[144] Seznec, A. TAGE-SC-L branch predictors. In *Proc. of the 4th Championship on Branch Prediction (`http://www.jilp.org/cbp2014/`)* (2014). Accessed online April 2018 from, `https://hal.inria.fr/hal-01086920/document`.

[145] Shahzad, R. K., and Lavesson, N. Veto-based malware detection. In *Proc. IEEE Int. Conf. on Availability, Reliability and Security (ARES)* (2012), pp. 47–54.

[146] Sheen, S., Anitha, R., and Sirisha, P. Malware detection by pruning of parallel ensembles using harmony search. *Pattern Recognition Letters 34*, 14 (2013).

[147] Shivakumar, P., and Jouppi, N. P. Cacti 3.0: An integrated cache timing, power, and area model, 2001. Technical Report 2001/2, Compaq Computer Corporation.

[148] Shokri, R., Stronati, M., and Shmatikov, V. Membership inference attacks against machine learning models. *arXiv preprint arXiv:1610.05820* (2016).

[149] Smutz, C., and Stavrou, A. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *Proc. Network and Distributed System Security Symposium (NDSS)* (2016).

[150] Sokolova, M., and Lapalme, G. A systematic analysis of performance measures for classification tasks. *Information Processing and Management 45*, 4 (jul 2009), 427–437.

[151] Stecklina, J., and Prescher, T. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).

[152] Summerville, A. Protect against the fastest-growing crime: cyber attacks, July 2017. Available online at: `https://www.cnbc.com/2017/07/25/stay-protected-from-the-uss-fastest-growing-crime-cyber-attacks.html`.

[153] Tang, A., Sethumadhavan, S., and Stolfo, S. J. Unsupervised anomaly-based malware detection using hardware features. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2014), Springer, pp. 109–129.

[154] Taram, M., Venkat, A., and Tullsen, D. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019).

[155] Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. Stealing machine learning models via prediction apis. In *USENIX Security* (2016).

[156] Trippel, C., Lustig, D., and Martonosi, M. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802* (2018).

[157] Trippel, C., Lustig, D., and Martonosi, M. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802* (2018).

[158] Tung, L. Linux meltdown patch: 'up to 800 percent cpu overhead', netflix tests show, Feb. 2018. ZDNet article: `https://www.zdnet.com/article/linux-meltdown-patch-up-to-800-percent-cpu-overhead-netflix-tests-show/`.

[159] Turner, P. Retpoline: a software construct for preventing branch-target-injection. `https://support.google.com/faqs/answer/7625886`, 2018.

[160] Van B., J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proc. of USENIX* (2018).

[161] van Schaik, S., Milburn, A., sterlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy (Oakland)* (May 2019).

[162] Vorobeychik, Y., and Li, B. Optimal randomized classification in adversarial settings. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems* (2014), International Foundation for Autonomous Agents and Multiagent Systems, pp. 485–492.

[163] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. Efficient software-based fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (New York, 1993), ACM Press, pp. 203–216.

[164] Wang, G., Chattopadhyay, S., Gotovchits, I., Mitra, T., and Roychoudhury, A. oo7: Low-overhead defense against spectre attacks via binary analysis. *arXiv preprint arXiv:1807.05843* (2018).

[165] Wang, K., Parekh, J., and Stolfo, S. Anagram: A content anomaly detector resistant to mimicry attack. In *Proc. of Recent Advances in Intrusion Detection (RAID)* (2006).

[166] Webroot. What is antivirus software? `https://www.webroot.com/us/en/resources/tips-articles/what-is-anti-virus-software`.

[167] Weisse, O., Van, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T., and Yarom, Y. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Tech. rep., 2018.

[168] Wilbur, K. C., and Zhu, Y. Click fraud. *Marketing Science 28*, 2 (2009), 293–308.

[169] Witten, I. H., and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[170] Wolpert, D. H. Stacked generalization. *Neural Networks 5* (1992), 241–259.

[171] Xu, W., Qi, Y., and Evans, D. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium* (2016).

[172] Yan, G., Brown, N., and Kong, D. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 41–61.

[173] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C., and Torrellas, J. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proc. of MICRO* (2018).

[174] Yarom, Y., and Falkner, K. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *Proc. of USENIX Security* (2014).

[175] Ye, Y., Chen, L., Wang, D., Li, T., Jiang, Q., and Zhao, M. SBMDS: an interpretable string based malware detection system using svm ensemble with bagging. *Journal in computer virology 5*, 4 (2009), 283–293.

[176] Yeh, T.-Y., and Patt, Y. N. Alternative implementations of two-level adaptive branch prediction. In *ACM SIGARCH Computer Architecture News* (1992), vol. 20, pp. 124–134.

[177] Yerima, S. Y., Sezer, S., and Muttik, I. High accuracy android malware detection using ensemble learning. *IET Information Security* (2015).

[178] You, I., and Yim, K. Malware obfuscation techniques: A brief survey. In *Proc. International Conference on Broadband, Wireless Computing, Communication and Applications* (2010), pp. 297–300.

[179] You, I., and Yim, K. Malware obfuscation techniques: A brief survey. In *Proc. International Conference on Broadband, Wireless Computing, Communications and Applications (BWCCA)* (2010), pp. 297–300.

[180] Zhang, B., Yin, J., Hao, J., Zhang, D., and Wang, S. Malicious codes detection based on ensemble learning. In *Lecture Notes in Computer Science* (2007), vol. 4610, Springer, pp. 468–477.

[181] Zhang, H., Yao, D. D., and Ramakrishnan, N. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2014), ASIA CCS '14, ACM, pp. 39–50.

[182] Zhang, H., Yao, D. D., and Ramakrishnan, N. Causality-based sensemaking of network traffic for android application security. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security* (New York, NY, USA, 2016), AISec '16, ACM, pp. 47–58.

[183] Zhang, M., and Sekar, R. Control flow integrity for cots binaries. In *USENIX Security Symposium* (2013), pp. 337–352.

[184] Zhou, B., Gupta, A., Jahanshahi, R., Egele, M., and Joshi, A. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (2018), ACM, pp. 457–468.